Gestão de Plataformas e APIs

Estratégia e discovery para product managers não técnicos

Casa do Código | alura

SHEILA CHANG

Sumário

- ISBN
- Prefácio por Joca Torres
- Gestão de plataforma, você sabe como fazer?
- PARTE 1 — Fundamentos para gestão de plataforma e API como produto
  - 1 O que é plataforma, afinal?
  - 2 Anatomia da plataforma
  - 3 Gestão do ciclo de vida
  - 4 Começando em um time ou plataforma
- PARTE 2 — Estratégia para plataforma e API
  - 5 Em busca de platform-market fit
  - 6 Problemas e soluções
  - 7 Segmento de clientes e canais
  - 8 Proposta de valor e casos de uso
  - 9 Métricas
  - 10 Estrutura de custos e fluxos de receita
- PARTE 3 — Discovery de plataforma e API
  - 11 Em busca de eficácia
  - 12 Empatia: Exploração das necessidades
  - 13 Definição: Entendimento dos problemas e necessidades
  - 14 Ideação: Levantamento de ideias de soluções
  - 15 Prototipagem: Criação de soluções
  - 16 Teste: Validação das hipóteses
  - 17 Como PMs conseguem testar APIs
  - 18 Da ideação ao teste de usabilidade
- PARTE 4 — Depois do discovery
  - 19 Em busca de…
  - 20 Juntando tudo
  - 21 Referências

ISBN

Impresso: 978-85-5519-383-5

Digital: 978-85-5519-384-2

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Higena, Sheila Chang
Gestão de plataformas e APIs : estratégia e discovery para product managers não técnicos / Sheila Chang Higena. -- 1. ed. -- São Paulo : AOVS Sistemas de Informática, 2024.

Bibliografia.
ISBN 978-85-5519-383-5

1. Administração de empresa 2. Interface de programas aplicativos (Software) 3. Interfaces de usuários (Sistema de computação) 4. Plataforma digital 5. Tecnologia I. Título.

24-226155 CDD-005.1068

Índices para catálogo sistemático:

1. Software : Desenvolvimento : Projetos : Gerenciamento : Ciência da computação 005.1068

Aline Graziele Benitez - Bibliotecária - CRB-1/3129

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Prefácio por Joca Torres

Em 2006, na Locaweb, tínhamos cinco produtos: a Hospedagem de Sites, o Email, a Revenda de Hospedagem de Sites, o Servidor Dedicado e estávamos lançando o Construtor de Sites. Todos os produtos ligados e derivados do produto original da Locaweb.

O ponto em comum de todos esses produtos é que em todos havia a necessidade de enviar emails. No produto de Email essa necessidade é óbvia. Nos outros produtos, todos eles estão relacionados a sites que, em alguns casos, podem necessitar enviar algum tipo de email (confirmação de login, recuperação de senha, detalhes do pedido etc.). Contudo, como em cada produto estávamos usando uma tecnologia diferente, em cada produto usávamos uma solução diferente para envio de emails.

Por volta de 2010 nos demos conta disso e pensamos que deveríamos ter uma única solução de envio de email que servisse para todos os produtos. Depois começamos a perceber que nossos produtos tinham vários outros "pedaços" que eram comuns. Autenticação e autorização, API, infraestrutura de deploy. Foi aí que percebemos a necessidade de criar um time que iria cuidar desses "pedaços" comuns dos produtos. Chamamos esse time de Foundation, e ele era formado somente por pessoas engenheiras. Depois de mais um ano com esse time rodando só com pessoas engenheiras, percebemos que precisávamos de um olhar de gestão de produto, para conectar o trabalho desse time com as necessidades dos outros times, com os problemas dos clientes e com os objetivos estratégicos da Locaweb.

Tudo isso para contar que levamos uns 6 anos para perceber e resolver algo que, se tivéssemos o livro que você tem em mãos agora, resolveríamos em questão de semanas. A Sheila, que trabalha com produtos digitais há mais de 20 anos, desde maio de 2021, passou a cuidar dos produtos que eram "pedaços" comuns dentro do iFood. Dentre os produtos que ela cuidou está a plataforma de

pedidos, um "pedaço" comum a mais de 100 aplicações. O que chamei aqui de "pedaço" comum tem o nome de plataforma, mais especificamente *plataforma de produto*, como a Sheila bem explica no capítulo 1, "O que é plataforma, afinal?", uma vez que plataforma é um termo bastante genérico e pode significar várias coisas. E, com as plataformas de produto, nasce o papel da pessoa *Product Manager de plataforma*, que também pode ser *Technical Product Manager* ou *API Product Manager*.

Se você é uma pessoa Product Manager que se deparou com a necessidade de gerir um produto de plataforma, ou se você está em um time de plataforma sem Product Manager e está querendo entender como fazer com que o trabalho do time esteja mais alinhado com as necessidades dos outros times e, consequentemente, da empresa e dos clientes, ou ainda, se você tem mais de um time de desenvolvimento de produto e está começando a observar uma certa duplicação em algumas coisas que esses times estão fazendo, então está na hora de você se aprofundar no tema de gestão de plataformas de produto, e a Sheila é a melhor guia para isso. Não só pela sua experiência em uma das maiores empresas de tecnologia do Brasil e, provavelmente, do mundo, como também pela sua capacidade de organizar seus aprendizados neste livro.

Ela vai falar sobre conceitos, ciclo de vida, estrutura de time, valor da plataforma para stakeholders, incluindo os stakeholders não técnicos, qual estratégia aplicar, que métricas acompanhar, *discovery*, ideação de hipóteses de solução, prototipação, teste de usabilidade, ou seja, como aplicar vários conceitos, princípios e ferramentas comuns da gestão de produtos que são *customer-facing* nesses produtos de plataforma, produtos utilizados pelos outros times de produtos para reduzir seu *time-to-market*, seus riscos e seus custos.

Enfim, aqui você vai encontrar tudo o que você precisa para estar pronto ou pronta para rodar com produtos de plataforma em poucas semanas, tendo a Sheila como guia, em vez de ter que investir meses ou anos no método de tentativa-e-erro, como foi o meu caso tentando aprender isso tudo por conta própria! Aproveite!!!

Joca Torres

Gestão de plataforma, você sabe como fazer?

Se sua resposta foi NÃO, é bem provável que você esteja passando pelo mesmo que eu passei.

Quando comecei a trabalhar com plataforma de produtos no iFood, pensei: por que me contrataram para gestão desse produto?

Na entrevista, eu disse que eu queria trabalhar com produtos “técnicos", como os que eu usava, Google Analytics, Shortcut (chamava Clubhouse), Productboard. Falei que eu gostaria de trabalhar na experiência do dev (DX) e mencionei o Jira, GitHub e GitLab. Fui contratada para fazer gestão do tal produto "técnico". Só teve um probleminha! A plataforma para a qual fui contratada para fazer a gestão estava longe, infinitamente distante, do que o que passou pela minha cabeça na entrevista, quando ouvi o termo “plataforma”. Nos primeiros dias, descobri que envolvia experiência do dev, mas era mais “técnico” do que imaginei, eram APIs.

Eu já trabalhava com produtos digitais havia mais de 20 anos, tinha atuado em diferentes setores e etapas do funil — financeiro, e-commerce, marketplace; aquisição, ativação, retenção e monetização. Devido a isso, a expectativa sobre meu trabalho era altíssima.

Já minha percepção era de que todo meu repertório, acumulado ao longo de anos, era inútil para este tipo de produto. Claro que me desesperei e me lembro de, pelo menos, cinco momentos em que eu quis desistir. Primeiro, expressei ao head de produto o quão inútil era meu conhecimento para esse tipo produto. Então, ele indicou o que eu deveria estudar de plataforma e a confiança de que eu conseguiria — nas palavras dele, “você é a pessoa certa para esse produto”.

Como eu já estava lá, resolvi encarar. Para isso, tive que estudar muito — e rápido!

Nos primeiros três meses, trabalhei no Faster, uma plataforma puramente de APIs. Achei difícil, duas vezes pensei em desistir, mas depois que entendi bem o produto, os usuários e necessidades deles, até que ficou tranquilo. Visão do produto e o planejamento do ciclo bem definidos, com *discovery* sólido e fluxo de *delivery* indo bem.

Fui convidada para fazer a gestão da Order Platform, a plataforma que processa os pedidos. Ela é o coração do iFood, no centro do ecossistema. Nela, o pedido é criado, tem os status atualizados ao longo do seu ciclo de vida e os dados são propagados a todos os serviços interessados. Há mais de 100 aplicações integradas a ela. A essas aplicações integram-se tantas outras que nem temos noção do tamanho da rede que formam, o que provavelmente faz da Order Platform a mais complexa plataforma do iFood.

Comecei a trabalhar nessa plataforma bem na época de planejamento para o próximo ciclo. Mas como planejar algo que eu nem conhecia? Para agravar a situação, eu fui a primeira PM do time. Nas *dailies*, era uma sucessão de termos que eu nunca tinha escutado. Eu não entendia nada! Por três vezes, pensei em desistir.

Ainda bem que continuei. O esforço me trouxe uma satisfação inigualável. Hoje, posso dizer que, de todos os produtos em que trabalhei, este foi o mais difícil, o que mais amei e o que mais me orgulha. Os resultados ficaram evidentes: time alinhado aos objetivos, sabia o que fazer no ciclo, valor aderente à estratégia do negócio e apoio da alta gestão.

Para quem é o livro

Se você é *Product Manager* e migrou recentemente da gestão de produto *customer-facing* (voltado para o usuário final) para plataforma e está perdido(a), este livro vai ajudar você nessa

transição. Também ajudará se você quer aprender sobre gestão de plataforma para abrir-se a um leque maior de oportunidades de trabalho como PM, este livro traz luz sobre os desafios que você vai encontrar e você será capaz de encarar uma nova oportunidade, dentro ou fora da empresa em que você trabalha.

Este livro é para pessoas que já atuam como *Product Manager* e que já têm algum domínio de técnicas de *discovery*, *delivery* e lançamento, mas devido ao fato de a plataforma se tratar de um produto técnico, estão com dificuldades de fazer a gestão — ou não sabem por onde começar e o que precisam aprender sobre o assunto.

Embora seja impossível falar de plataforma sem aprofundar em abordagens técnicas, este livro não se destina às pessoas de engenharia, que desenvolvem ou consomem APIs, tampouco ajudará em decisões técnicas dessa área.

O que você vai encontrar neste livro

Se você faz gestão de plataforma, já deve ter se perguntado:

- Qual valor que plataforma entrega? Como provar o valor aos *stakeholders* de negócio?
- Quais são as métricas de sucesso? Como conectar aos objetivos do negócio?
- Como priorizar diante de tantas demandas de diferentes *stakeholders*?
- Como lidar com a dependência entre produtos e times?
- Como fazer *discovery*?
- O que é preciso saber sobre a tecnologia?
- Como definir o domínio: até onde o produto vai e depois disso é responsabilidade de outro produto?

Neste livro, você encontrará respostas para essas e outras perguntas. Verá que a base dos conceitos de gestão de produtos é aplicável a qualquer tipo de produto: xampu, carro ou produto digital, inclusive, plataforma.

Minha teoria é de que precisamos compreender as características inerentes para a existência do produto e o que cada estágio do ciclo de vida do produto exige de estratégia e tático.

Com isso em mente é que consegui (junto com o time, óbvio!) lançar a nova Order Plataform. Por esse motivo, recebi a "demanda" de compartilhar dentro do iFood meu conhecimento e **como apliquei práticas comuns de gestão de produtos na gestão de plataforma**, o que me deu a ideia de ir além, escrever um livro.

A ideia se materializou neste livro que você tem em mãos. Ele dá uma visão completa do processo, teoria e prática, para planejar, elaborar o design e gerenciar o desenvolvimento de plataforma, desde o valor até o suporte ao desenvolvedor, seja para uma nova plataforma ou melhorias de uma existente.

Parte 1 — Fundamentos para gestão de plataforma e API como produto

Você verá o lado técnico, as características de uma plataforma (microsserviços, backend e API síncrona e assíncrona); o papel de cada profissional do time, incluindo Designer de API, um papel pouco difundido, e o que *Platform Product Managers* devem procurar fazer nos primeiros 90 dias. Além disso, o ciclo de vida da gestão de plataforma e o que devemos nos atentar em cada estágio, que são mais explorados ao longo do livro.

Parte 2 — Estratégia para plataforma e API

De posse dos fundamentos, a segunda parte apresenta o geral da estratégia, com as etapas que você precisa seguir para iniciar a criação de uma plataforma, que abrange o valor, métricas de sucesso e compreensão dos casos de uso. Ao término da parte 2,

você conseguirá definir a estratégia da plataforma, usando o *Lean Canvas*, que adaptei para plataforma.

Parte 3 — Discovery de plataforma e API

Depois da estratégia, é hora de validar as hipóteses enumeradas no *Lean Canvas*. Você saberá como se beneficiar do Design Thinking para criar uma plataforma centrada nos usuários e como utilizei ferramentas conhecidas nas práticas de produto, como Matriz CSD, User Story Mapping, Wireframe. Você vai conhecer conceitos e boas práticas de design API REST, como testar, criar protótipo e aplicar teste de usabilidade.

Parte 4 — Depois do discovery

Para encerrar a jornada, vem um apanhado geral das etapas seguintes — Delivery, Lançamento e Monitoramento — e reflexões acerca dos desafios de *Platform Product Managers*, porque esses desafios ocorrem e como você pode lidar com eles.

Depois de aprender o necessário para a gestão de um produto tão técnico, você sentirá mais segurança nas suas decisões e saberá o que fazer, como fazer e quando fazer.

Agradecimentos

Este livro é o resultado de uma jornada compartilhada, que só foi possível graças à generosidade, expertise e suporte de pessoas fora do comum que me acompanharam ao longo do caminho.

Primeiramente, sem a compreensão e ajuda do time da Order Platform eu nem teria conseguido desempenhar meu papel de PM.

Um agradecimento especial pela paciência de me ensinar o techniquês ao Augusto Andraus, Danilo Souza, Lucas Moretti, Luiz Boaventura, Michel Nagme, Rodrigo Yasuda e Vinícius Pereira. Cada um de vocês asseguraram que nossas práticas de discovery gerassem resultados. Vocês não só me ajudaram a ser um PM de plataforma, como também foram fundamentais em cada página deste livro.

Não posso deixar de agradecer de coração ao Alex Zanetti e Flávio Stecca pela disposição em ouvir, discutir soluções e orientar o time. Com vocês, aprendi não apenas as tecnicidades, mas principalmente sobre o negócio e complexidades do iFood.

Um agradecimento especial ao Eduardo Olímpio, a pessoa que acreditou em mim, me contratou no iFood e depositou confiança de que eu seria capaz de aprender sobre o tema e conseguiria fazer gestão de plataforma. A pessoa que não me deixou desistir e iluminou meu caminho nessa empreitada. Sem sua orientação, eu não teria me tornado Platform Product Manager e este livro jamais seria imaginado.

A todos vocês, minha sincera gratidão. Vocês são parte da história por trás deste livro. Obrigada por cada contribuição e pelo privilégio

de aprender e crescer ao lado de cada um de vocês.

PARTE 1 — Fundamentos para gestão de plataforma e API como produto

Capítulo 1
O que é plataforma, afinal?

Instagram, Mercado Livre, Uber, Salesforce, AWS, DropBox, Zoom, Android. Todas são plataformas, e tão diferentes. Assim como no mundo físico há diversas plataformas (plataforma de petróleo, de embarque, de lançamento de foguete e até sapato plataforma), também há diversas no mundo digital (plataforma de software, de engenharia, de e-commerce, de API), mas com uma grande diferença: podemos encontrar diferentes tipos de plataformas com o mesmo nome; e o inverso também é verdadeiro, nomes diferentes para o mesmo tipo de plataforma. É isso o que nos confunde.

Neste capítulo, vamos entender algumas delas: as plataformas de negócios, de computação, de produto e de tecnologia. Para ilustrar esses tipos de plataforma, vejamos um recorte específico e (muito) simplificado de uma das operações do iFood: pedidos de refeição.

1.1 Negócio de plataforma

Antes de tudo, o **negócio de plataforma (*platform business*) é um modelo de negócio**. Ganhou notoriedade pelo efeito de rede, em que, quanto mais usuários houver, mais valor eles recebem da plataforma como retorno, o que atrai ainda mais usuários.

Este tipo de plataforma atende a dois ou mais segmentos de clientes (*multi-sided market*) que possuem interesses mútuos — *ou seria complementares?* —, em que o valor da plataforma é viabilizar o encontro entre demanda e oferta, quem quer comprar e quem quer vender. Como o iFood, que, enquanto negócio de plataforma, intermedia a interação entre três segmentos de clientes:

- Consumidor: a pessoa que faz o pedido de refeição e paga por ele (demanda);
- Restaurante: o estabelecimento que comercializa a refeição e recebe o pagamento (oferta) e torna-se consumidor quando contrata serviço de entrega (demanda);
- Entregador: a pessoa que pega o pedido no restaurante, o entrega ao consumidor e recebe o pagamento (oferta).

Percebeu que, a depender do momento no fluxo, quem oferta pode gerar demanda? Como vimos, em torno do pedido, o restaurante atua como fornecedor da oferta (refeição) e como consumidor de outra oferta (serviço de entrega). Os papéis podem se inverter também. Em um momento, o entregador é fornecedor; em outro, ele pode ser consumidor, gerando demanda ao restaurante que, por sua vez, vai gerar demanda a outro entregador.

Uma característica importante a observar é que o produto ou serviço comercializado não pertence ao iFood, pertence aos usuários — a refeição é do restaurante, a moto é do entregador. O iFood "apenas" viabiliza as interações entre as partes. Então, qual a vantagem para os três perfis de usuários?

O iFood fornece a infraestrutura e a tecnologia que tornam as interações possíveis. Agrega valor pela comodidade de ter no mesmo lugar uma grande variedade de ofertas, com filtros e recomendações que agilizam a busca por restaurantes próximos ou por comidas que o consumidor provavelmente vai gostar, o que o leva a conhecer novos restaurantes. Consequentemente, a demanda para os restaurantes e entregadores aumenta, o que leva ao aumento de receita para ambos.

Olhando o caminho feliz, “apenas” viabilizar as interações entre usuários parece simples e fácil. A verdade é que exige uma operação parruda e, claro, muitas tecnologias! A começar pela existência de três apps (para atender às necessidades de cada segmento de cliente) que dependem das plataformas de computação.

1.2 Plataforma de computação

Qualquer aplicativo mobile depende das **plataformas de computação** (*computing platforms*) como o Google Android e o Apple iOS, sistemas operacionais que oferecem um conjunto de funcionalidades para o funcionamento dos celulares.

A característica desse tipo de plataforma é a constante expansão de suas funcionalidades conforme cada novo app é integrado. Empresas podem desenvolver seus aplicativos e acrescentar novos casos de uso aos celulares, isto é, estender as capacidades do sistema operacional, tornando-o mais valioso.

Dessa forma, os apps são complementares às plataformas. Há uma grande variedade de complementos disponíveis que as torna mais desejáveis. Isso, inclusive, pode ser o diferencial na decisão de compra entre um celular Android e iPhone, por exemplo. A pessoa pode optar por um, em detrimento do outro, porque o app preferido dela existe para uma plataforma e não para a outra.

Também conhecida como **plataforma de indústria** — não no sentido industrial e, sim, no sentido de mercado —, atende a dois segmentos de clientes (mercado bilateral — *bilateral market*): desenvolvedores de apps e usuários dos apps.

Assim, as plataformas Android e iOS têm o iFood como cliente de um lado e, do outro, os usuários dos celulares, que antes de serem clientes do iFood, são clientes dessas plataformas. Em contrapartida,

os apps iFood adicionam a capacidade de compra e venda de refeições aos celulares Android e iOS.

1.3 Plataforma de produto

Modelos de negócio como o do iFood exigem múltiplos produtos: apps mobile e web para atender a cada tipo de perfil de usuário e tantos outros produtos internos para suportar a operação. Além disso, as empresas almejam crescimento e apostam em novos nichos e modelos de negócios, ofertando mais produtos no mercado.

Figura 1.1: Apps iFood. Fonte: Google Play Store

Chega um momento em que a tecnologia usada até então não suporta mais a demanda em crescimento rápido e contínuo e/ou percebe-se que diferentes times desenvolveram a mesma

funcionalidade, o que é um grande desperdício. São sinais de que é hora de "plataformizar" e, assim, a **plataforma de produto (*product platform*) entra como estratégia** — a protagonista deste livro.

Desde a década de 1970, a plataforma de produtos é usada como estratégia por diversos setores. A Volkswagen é um exemplo clássico. Nos anos de 1980, a empresa queria escalar o negócio, alcançar outros nichos de mercado. Para isso, precisava de novos modelos de carros.

Mas como penetrar esses nichos rapidamente? Precisavam de mais eficiência na produção. Desenvolveram, então, a primeira plataforma de carros, com estrutura modular e padronizada, que possibilitou o uso da mesma base em modelos de carros variados.

Em princípio, o objetivo era aumentar a capacidade interna de criação e produção para acelerar o *time-to-market* (tempo para disponibilizar para venda), ou seja, alavancar vantagem competitiva. Segundo a Volkswagen, a estratégia levou também à redução de custos por unidade.

Bruce Cameron, professor de Sistemas de Engenharia no MIT, conduziu uma pesquisa por 8 anos em 32 empresas que usam plataforma de produto para desenvolver produtos físicos (carro, satélite, avião). Em um webinar (MIT SLOAN EXECUTIVE EDUCATION, 2015), ele conta sobre essa pesquisa e aponta três benefícios centrais para o negócio obtidos por meio da estratégia de plataforma:

1. **Redução de *time-to-market***: há menos a ser desenvolvido para novos produtos entrarem no mercado; a empresa consegue responder mais rápido quando percebe oportunidades em novos nichos;
2. **Redução de custos**: elimina desperdício de tempo em aprendizado e dinheiro para desenvolver um recurso que já existe na empresa;

3. **Redução de riscos**: quanto mais se usa a plataforma, mais problemas podem ser identificados e, portanto, corrigidos, o que aumenta a qualidade.

Ao longo dos anos, a Volkswagen desenvolveu várias plataformas. Atualmente, a empresa também oferta plataformas no mercado para que outras empresas possam desenvolver automóveis, pela unidade de negócio Volkswagen Platform Business (https://www.vw-platform-business.com/en.html). Aliás, a Volks não está sozinha nesse mercado. Existe até plataforma *open source*! A TABBY EVO, uma plataforma de hardware de código aberto para veículos elétricos projetada pela OpenMotors (https://www.openmotors.co/evplatform/).

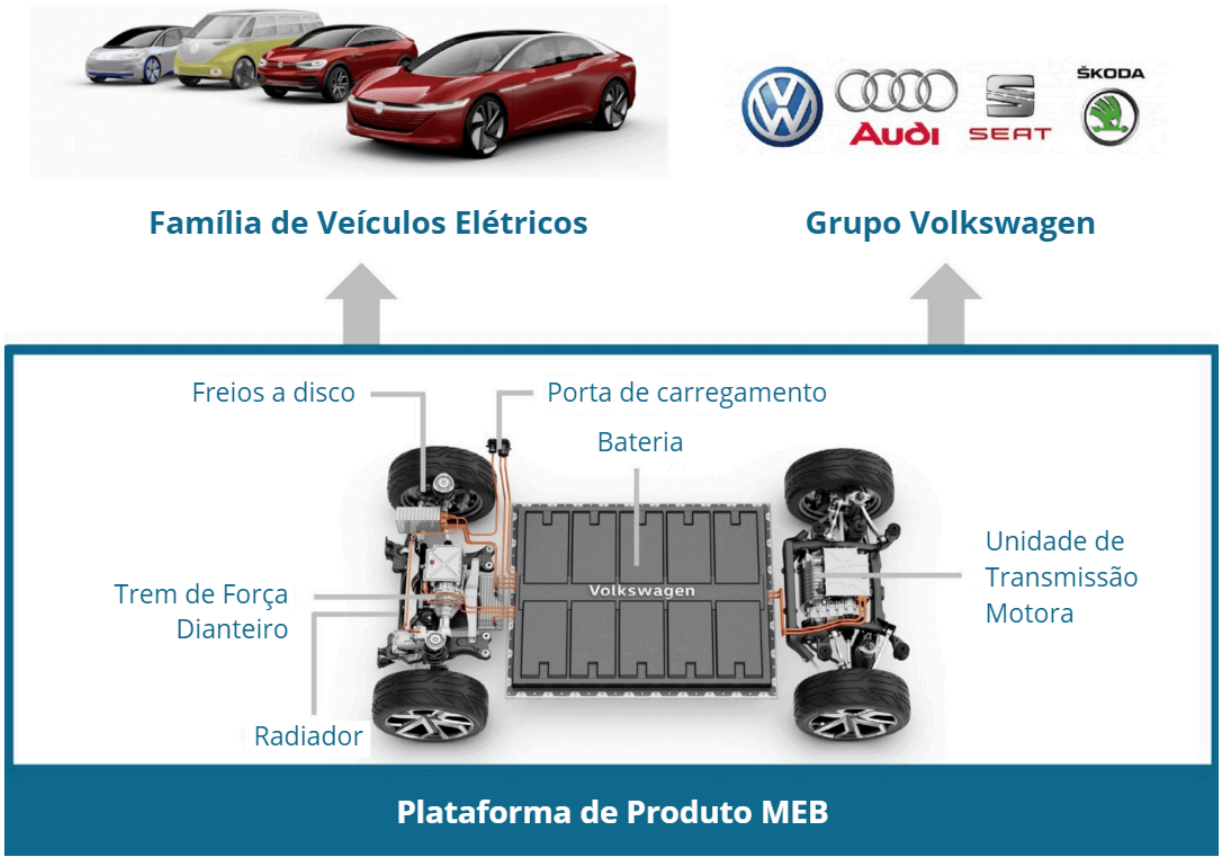


Figura 1.2: Plataforma Volkswagem. Imagem adaptada de: A Strategist's Guide to Platform Thinking (KOLOSSOVSKI, 2019)

Assim como para produtos físicos, o conceito que define plataforma de produto digital é o compartilhamento de recursos já existentes na empresa, como componentes, design e infraestrutura, elaborados de modo que possam ser reutilizados internamente em diferentes produtos — *building blocks* — , assim como blocos de montar infantil em que peças básicas e criatividade são capazes de gerar infinitas formas.

A exemplo da Volkswagen, em algum momento, a empresa pode optar por disponibilizar a plataforma no mercado, como fez a AWS. Mas, historicamente, o conceito nasceu da necessidade interna de a empresa alavancar a eficiência no desenvolvimento de novos produtos, com o objetivo primordial de acelerar o lançamento no mercado.

De volta ao iFood, para um pedido ocorrer com sucesso, além dos apps para os três perfis de clientes, há várias (sim, várias!) plataformas de produto envolvidas.

Uma delas é a Order Platform, a plataforma que processa os pedidos. Ela está no centro do ecossistema e arrisco a dizer que é a mais complexa, devido à quantidade de interação que formam uma teia de relacionamentos entre aplicações — mais de 60 milhões de pedidos que impactam mais de 140 aplicações de mais de 30 squads.

A plataforma recebe os dados do pedido e é responsável por criá-lo, atualizá-lo ao longo do ciclo de vida e propagá-lo a todos os serviços interessados até o pedido chegar nas mãos do consumidor.

É assim que, junto com outras plataformas, o restaurante fica sabendo que chegou um novo pedido; que o entregador é acionado para a entrega; que o consumidor sabe em qual etapa da jornada está o pedido; que as pessoas de atendimento conseguem fornecer informações e suporte; que o restaurante e o entregador recebem o repasse de pagamentos, entre outras coisas.

Há três checkouts conectados à Order Platform. Se cada um deles tivesse que desenvolver o próprio "processador" de pedidos, seria um caos analisar uma métrica simples, mas de suma importância, como o total de pedidos em determinado mês. Este é um caso típico em que a estratégia de plataforma para uso interno mostra o seu valor, mas o assunto não acaba por aqui. Para suportar os aplicativos/produtos, a plataforma de produto depende da plataforma de tecnologia.

1.4 Plataforma de tecnologia

Desde um simples site até a plataforma mais complexa do planeta dependem de infraestrutura, como servidores e banco de dados. A plataforma de tecnologia (*technology platform*) é onde se hospeda tudo o que é necessário para a execução e o acesso — código, arquivos, dados, configurações.

Para entender a importância da plataforma de tecnologia, imagine você com fome. Provavelmente, você tem pressa e seu humor não está dos melhores. Você abre o app do iFood, escolhe a sua comida preferida e, na hora de concluir o pedido, o sistema fica carregando eternamente. A Order Platform está indisponível, portanto não processa o seu pedido. O que você sente? Raiva? Ódio mortal?

Além de você, há inúmeros consumidores, restaurantes e entregadores expondo toda a ira deles nos canais de atendimento, nas redes sociais, no Reclame Aqui. Por isso, além de alta **performance**, as plataformas de produtos precisam de altíssima **disponibilidade** para se manterem acessíveis a dezenas ou, quem sabe, centenas de aplicações e de produtos que derivam delas.

Por conta disso, a moda é — pelo menos, por enquanto — usar plataformas de tecnologia de infraestrutura em nuvem (*cloud*), como Amazon Web Services (AWS) ou Google Cloud. A computação em

nuvem é distribuída em múltiplos servidores que atuam em conjunto, o que os torna mais tolerantes a falhas do que os servidores comuns, mesmo que dedicados.

Se um determinado servidor fica indisponível, o tráfego é automaticamente direcionado para outro, o que resulta em maior disponibilidade. A carga de tráfego é balanceada entre os servidores, o que garante performance.

É Black Friday e vai bombar de acessos e pedidos? Sem crise. A tecnologia em nuvem possibilita a **escala** de banda, armazenamento e capacidade computacional. Tudo isso se traduz em melhor experiência para o usuário final.

1.5 Juntando tudo

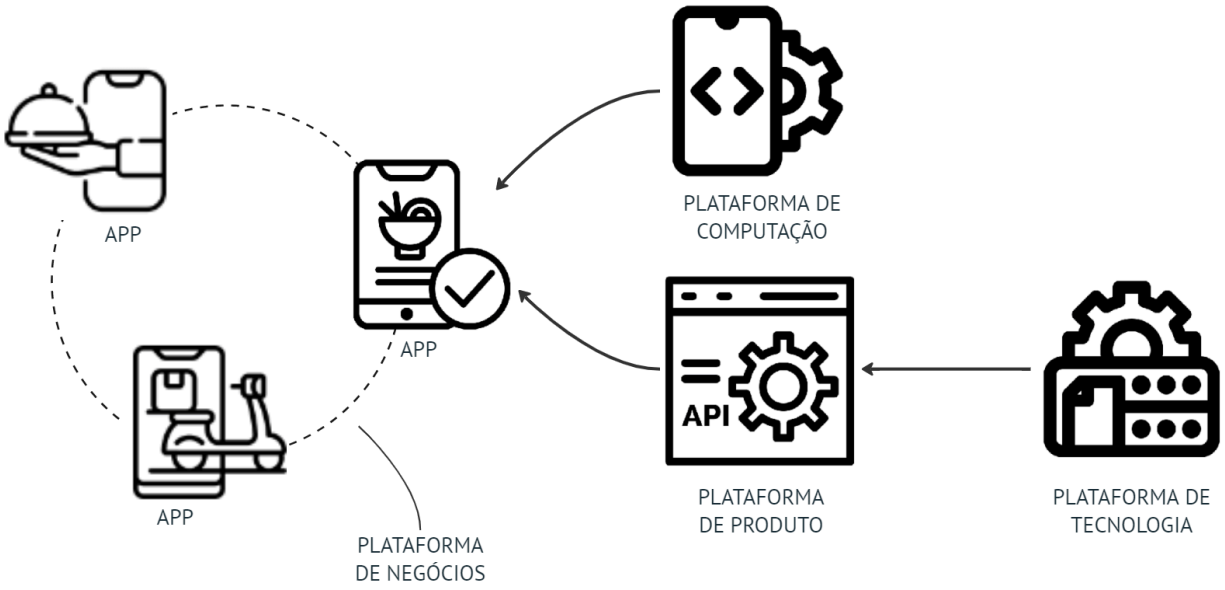


Figura 1.3: Ecossistema de plataformas

Geralmente, o negócio de plataforma é composto por apps específicos para cada perfil de usuários para viabilizar a interação entre as partes. Estes apps dependem da plataforma de

computação. Quando a empresa já tem determinada funcionalidade plataformizada, ela pode ser adicionada no aplicativo, tornando-se dependente também da plataforma de produto, que, por sua vez, depende da plataforma de tecnologia.

Confesso que tive de optar pelas nomenclaturas que vimos até aqui com base em referências clássicas ou nas que considero mais confiáveis e, talvez, você tenha percebido certa incoerência. E tem mesmo!

Primeira incoerência: a plataforma de computação atende a dois segmentos de clientes que produzem o efeito de rede. Usuários de celulares Android e iOS atraem desenvolvedores; desenvolvedores criam apps; estes apps atraem usuários; usuários atraem ainda mais desenvolvedores. Pois é, este também é um negócio de plataforma.

Segunda incoerência: você reparou que a plataforma de computação também é a fundação para criar e executar aplicações (produtos), o que faz dela uma plataforma de produto?

Minha intenção está longe de determinar um nome que seja usado mundo afora para os tipos de plataformas apresentados. O objetivo é citá-las ao longo do livro sem ter que explicar o conceito todas as vezes para especificar a qual delas estou me referindo.

Para PMs de plataforma, a plataforma *é* o produto. Então, para evitar confusão, convencionei o uso de certos termos:

- **App** para me referir a produtos *customer-facing*, seja aplicativos web ou mobile;
- **Plataforma** quando trato dos assuntos acerca da plataforma de produto;
- **Produto** expressa conceitos genéricos que englobam apps, produtos e plataformas.

Agora que estamos "falando a mesma língua", vamos conhecer as características da plataforma de produto.

Se você está com curiosidade sobre como cheguei à conclusão do nome "plataforma de produtos", veja o post *Plataforma de Produtos: que plataforma é essa?*, que escrevi no LinkedIn. Acesse: https://bit.ly/ppdefinicao.

Capítulo 2
Anatomia da plataforma

Produtos carregam consigo características estabelecidas pela categoria à qual pertencem. Uma calça jeans, por exemplo, deve seguir certas condições predeterminadas que a define como tal. O formato define que é calça; o tecido e a composição definem que é jeans. Do contrário, ela pode se tornar uma calça de moletom ou, quem sabe, uma camiseta.

Além disso, a calça jeans deve seguir padrões de qualidade que garantam que ela não se deteriorará após algumas horas de uso, e de segurança, que impeçam que a pessoa se machuque com a peça, por exemplo.

Como PM da calça jeans, você deve conhecer as características do seu produto e responsabilizar pela qualidade e as consequências do uso dele. Da mesma forma, como PM de plataforma, o primeiro passo é aprender sobre as características estabelecidas pela categoria "plataforma de produto" para entender o potencial e as limitações que a plataforma pode ter.

Por esse motivo, separei este capítulo para trazer o "techniquês", que amedronta PMs que não possuem histórico como desenvolvedores ou na engenharia de software. Ninguém sairá desta como cientista da computação, vamos ver apenas o suficiente para fazer a gestão da plataforma.

Vejamos, então, a Order Platform de modo simplificado. A plataforma é composta por duas partes. Para efeito didático, chamarei uma de plataforma de serviço e a outra de plataforma de dados.

1. **Plataforma de serviço**: aciona operações dentro do iFood ao longo do ciclo de vida do pedido, assim o restaurante é avisado

de que chegou um pedido e é possível alocar uma pessoa para a entrega, além de outras atividades para que o pedido ocorra com sucesso.

2. **Plataforma de dados**: onde os dados são armazenados para análise histórica, fornecer *insights* e predições, treinar modelos de *machine learning* e alimentar *dashboards* de negócio.

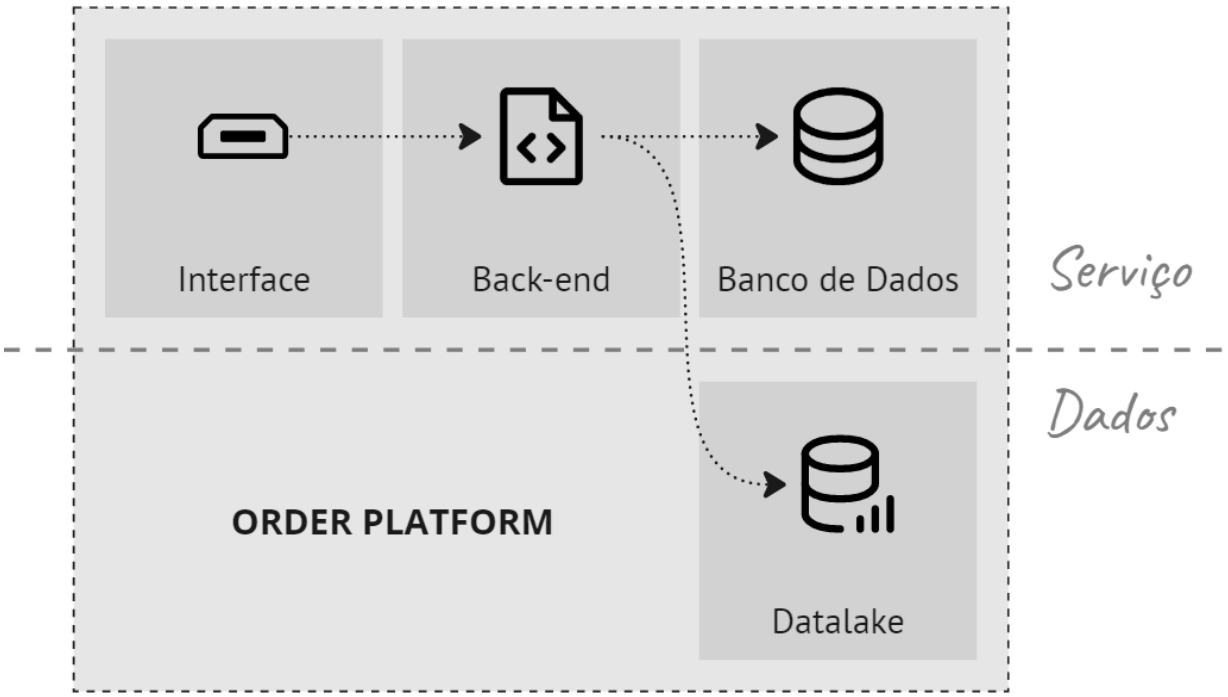


Figura 2.1: Estrutura da Order Platform.

A plataforma de serviço segue o estilo de **arquitetura de microsserviços** baseada em nuvem (**instância**), composta por:

- **Interfaces assíncronas e síncronas**: a forma de interagir com os demais serviços;
- **Back-end**: os recursos de computação que executam as regras de negócio;
- **Banco de dados**: onde são persistidos (salvos) os dados *realtime* que viabilizam a operação do negócio — por isso chamados de dados operacionais ou transacionais.

Estes dados passam pelo processo de ETL (*extract, transform, load*), isto é, são extraídos, transformados em visão temporal e agregada dos fatos do negócio e carregados no *datalake* para análise histórica (chamados de dados analíticos), formando a plataforma de dados. Não vou abordar a plataforma de dados, porque os problemas, soluções e usuários são diferentes aos do microsserviços, o que renderia outro livro — quem sabe o escrevo no futuro?

2.1 Microsserviços

No geral, plataformas são construídas usando arquitetura de microsserviços, assim chamados pois são limitados a um único propósito, a exemplo da Order Platform, que só processa pedidos. Ela não processa pagamentos, não tem responsabilidade pela conta dos usuários ou pelo raio logístico de entregas.

Dessa forma, para viabilizar uma operação como a do iFood, são necessários inúmeros microsserviços e, para a magia acontecer, os serviços precisam interagir entre si, caso contrário, como a Order Plataform saberia que um cliente quer fazer um pedido? O *checkout* é que sabe os itens que o cliente colocou no carrinho e os valores, então precisa passar essa informação para a Order Plataform criar o pedido e, assim, disparar a operação dentro do iFood. Essa comunicação entre as plataformas acontece por meio de uma *interface*.

2.2 Interface

Assim como o usuário final utiliza a interface visual para interagir no app mobile ou web, os códigos das aplicações, front-end ou back-end, precisam de uma interface para facilitar a interação com a

plataforma, mais precisamente a Interface de Programação de Aplicações (*Application Programming Interface*), popularmente conhecida como **API**.

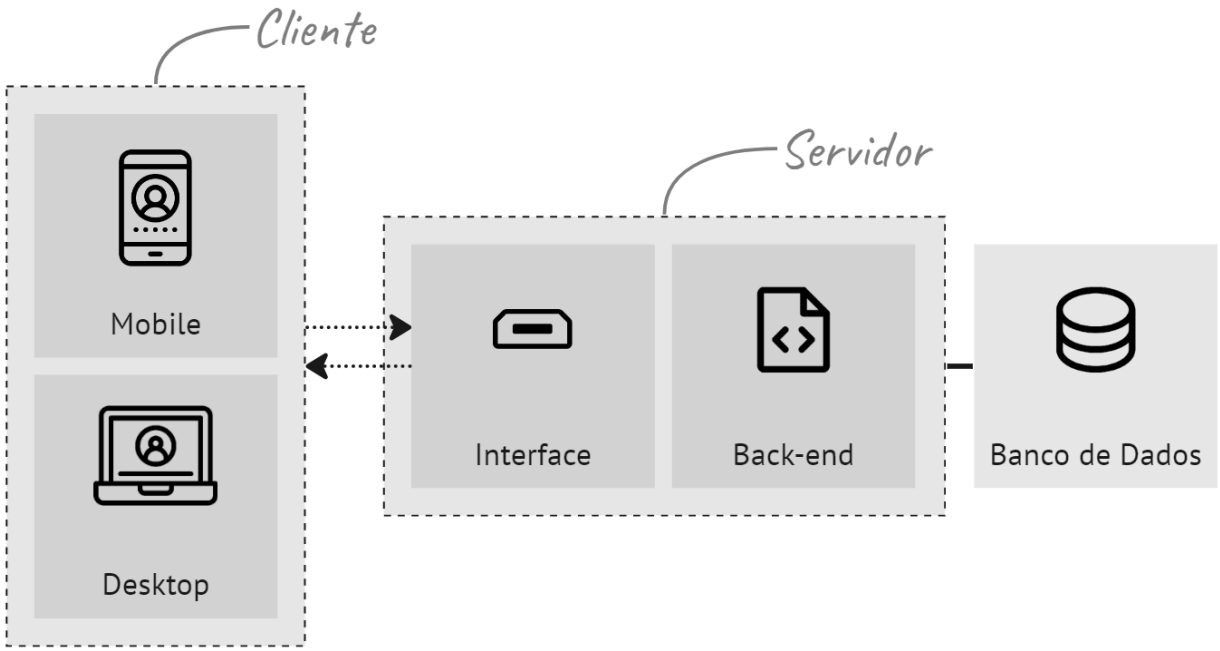


Figura 2.2: Cliente — API — Servidor.

Por exemplo, não é possível registrar uma conta de usuário no front-end. As informações do usuário devem ser validadas e armazenadas no banco de dados para que o acesso ao conteúdo seja liberado ao usuário. Então, quando o usuário faz login, o app (via API) chama o serviço de autenticação, que verifica se a conta existe no banco de dados e se as informações que o usuário preencheu correspondem aos dados que autorizam o acesso.

Também via API, o serviço devolve a resposta que libera o acesso ao usuário ou retorna um erro, como conta inexistente ou senha incorreta. O front-end exibe a mensagem personalizada ao usuário ou aquelas mensagens estranhas como *403 Forbidden* — ou, pior ainda, nada! Este é um caso típico em que a experiência do usuário final está fora do controle do time da plataforma.

Por outro lado, se o erro for algo como *503 Service Unavailable*, significa que o serviço está sobrecarregado ou indisponível, e o impacto da má experiência proporcionada ao usuário é causado pela plataforma. No capítulo *Como PMs conseguem testar APIs*, veremos mais sobre erros.

Tecnicamente, as APIs são categorizadas em:

- **API Privada**: é disponibilizada apenas internamente para interações entre sistemas da empresa.
- **API Pública**: é disponibilizada no mercado para uso fora da empresa e pode ter alguma forma de monetização, como a plataforma Google Maps, ou pode ser gratuita com a opção de logar com os dados do Facebook ou do Google, por exemplo.
- **API de Parceiros**: é semelhante à API pública, com a diferença de que o acesso externo é restrito aos parceiros da empresa.

Vale saber que essas categorias se referem ao público que a usa (interno, externo ou parceiro), não ao conteúdo que ela oferece ou aos apps que a utilizam.

A pesquisa *State of API Integration 2021 Report* (CLOUD ELEMENTS, 2021) mostra que a maioria das empresas (70%) usam APIs privadas; 44% usam APIs públicas e 32,3% usam APIs de parceiros. Na visão deles, o alto número de APIs privadas se deve ao aumento na adoção de microsserviços para lidar com necessidades internas e para desenvolver produtos *customer-facing*.

Um ano depois, a Postman chegou à mesma conclusão quanto a APIs privadas em uma pesquisa com 37 mil respondentes. A publicação dos resultados, *2022 State of the API Report* (POSTMAN, 2022), mostra que 58% das empresas usam APIs privadas; as APIs de parceiros ficam em segundo lugar (27%), e, por último, as APIs públicas (16%).

Um ponto a se observar é que o termo API é comumente usado ao se referir ao padrão REST (ou RESTful). Entretanto, o significado de

API é mais amplo, se refere a qualquer tipo de interface de comunicação. REST é apenas uma das arquiteturas possíveis de viabilizar a comunicação entre sistemas. Também é comum referenciar API como sendo a plataforma e vice-versa, mas tecnicamente, a interface é a parte da plataforma.

As APIs podem se comunicar de forma **síncrona**, o que significa que é necessário aguardar a resposta às requisições para que a operação continue; ou **assíncrona**, em que as respostas podem vir posteriormente, assim as operações são processadas continuamente, uma após a outra, sem necessidade de aguardar a resposta.

Comunicação síncrona

A arquitetura REST (*Representational State Transfer*) é a mais usada neste tipo de comunicação. A troca de informação ocorre por meio de protocolo HTTP — o padrão comum da internet — seguindo o modelo *request/response*.

Nesse estilo de arquitetura, de um lado há o sistema cliente (*client*), que faz a requisição (*request*). Do outro lado, o sistema servidor (*server*) processa e responde (*response*) a requisição ao sistema cliente. O sistema cliente aguarda a resposta para prosseguir com a operação.

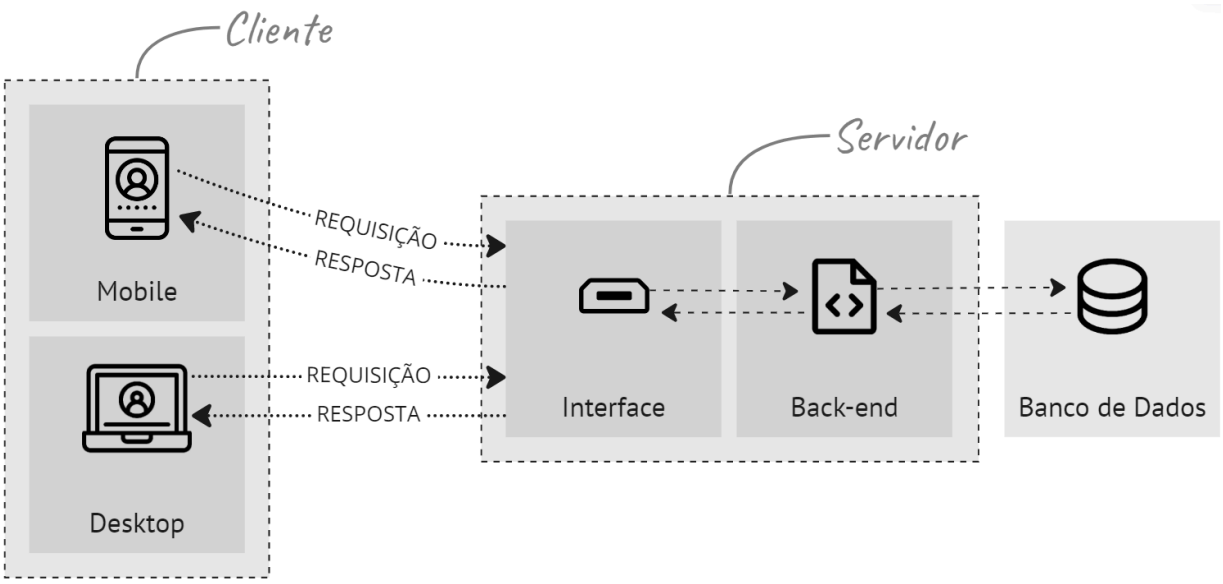


Figura 2.3: Requisição — Resposta.

Talvez você conheça essa arquitetura como RESTful; no entanto, REST e RESTful são complementares. Em poucas palavras, REST é um conjunto de regras e RESTful é a aplicação prática dessas regras ao construir APIs. Se RESTful segue obrigatoriamente as regras REST, pela lógica e segundo a Amazon, "*você pode usar os termos API REST e API RESTful de forma intercambiável*" (AWS, c2024).

Nenhum sistema tem recursos infinitos. A requisição síncrona ocupa recursos do sistema cliente e do sistema servidor durante todo o tempo do processamento. Isso significa que, se a plataforma suporta 100 requisições ao mesmo tempo, ao chegar mais uma, esta será perdida, assim como as demais que chegarem enquanto as 100 requisições estiverem em processamento.

Ao buscar uma informação em um banco de dados que leva 30 segundos para responder, a capacidade de processamento para outros usos ficará indisponível durante os 30 segundos em que o banco de dados busca o resultado. Ao chamar um serviço que está indisponível, ocorre falha na resposta. É necessário enviar mais uma requisição e, para isso, pode-se definir políticas de retentativa (*retry*) que garantam que a chamada acontecerá em algum momento.

A vantagem da API síncrona é que os sistemas clientes que enviaram as 100 requisições obterão os resultados esperados de imediato para seguir em frente.

Considere o processo de autenticação. Nesse cenário, é necessário aguardar uma resposta. Seria imprudente enviar a requisição para login do usuário, liberar o acesso e, como na comunicação assíncrona, tempos depois responder que a senha está errada. Em um caso como este, uma pessoa qualquer poderia usar o seu CPF para entrar na sua conta bancária e, só depois de muitos PIXs, o banco proibiria o acesso dela. Já pensou? Este é um caso em que esperar pela resposta é imprescindível e inquestionável.

Comunicação assíncrona

Também conhecida como *Event-Driven Architecture* (EDA), a comunicação assíncrona segue o modelo *publisher/subscriber* (pub/sub), ou *producer/consumer*, na qual a interação ocorre por meio de tópicos.

A comunicação assíncrona usa estruturas de fila de mensagens, em que, de um lado o produtor (*producer*) captura ou gera eventos e publica-os em um tópico; do outro, o consumidor (*consumer*) assina o tópico para receber os eventos e processá-los.

A comunicação pode ser intermediada por um *broker*, que organiza os dados nos tópicos, filtra-os, balanceia o volume e roteia as mensagens aos consumidores.

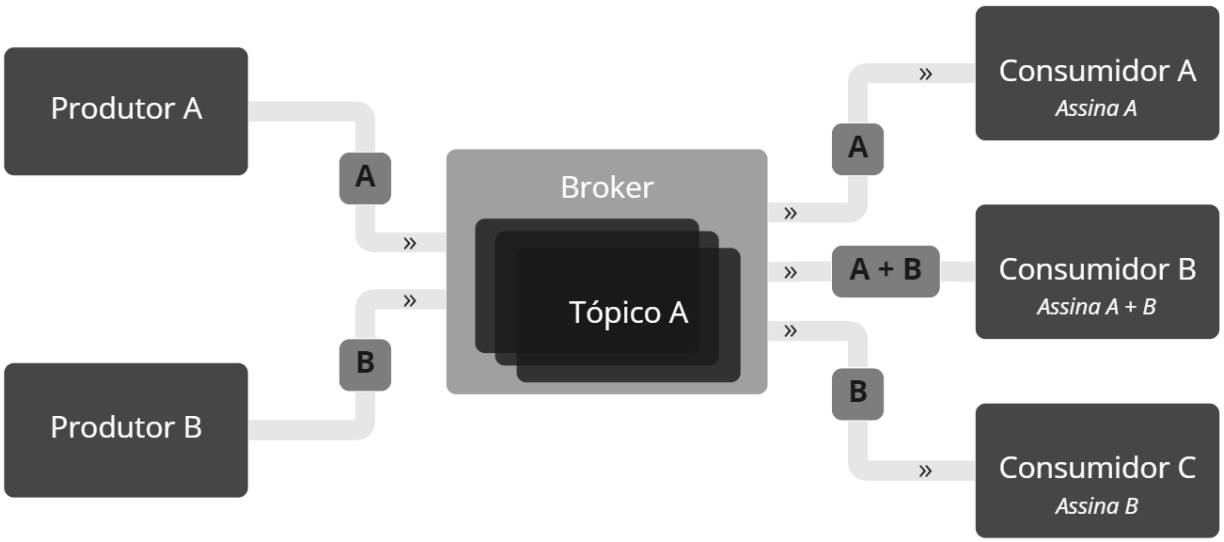


Figura 2.4: Produtor — Broker — Consumidor.

Uma vez que a mensagem é publicada no tópico, o sistema produtor não precisa esperar uma resposta imediata, então processa outras tarefas. Isso evita gargalos e melhora a escalabilidade.

Este método de comunicação tem a capacidade de atender a vários consumidores, que recebem cópias da mensagem que foi enviada uma única vez pelo serviço produtor. Novos consumidores podem ser adicionados ou removidos da lista de assinantes, sem necessidade de alterar o serviço produtor e sem impactar o desempenho do sistema.

Qualquer mensagem publicada em um tópico é imediatamente recebida por todos os assinantes (*subscribers*) do tópico. Se o assinante estiver indisponível, as mensagens são acumuladas na fila, prontas para consumo assim que o serviço consumidor estabilizar, garantindo que nenhum dado seja perdido. Dessa forma, o sistema produtor não depende da disponibilidade dos consumidores para se manter funcional.

No caso da Order Platform, ela cria o pedido e o envia aos sistemas consumidores, mas não espera resposta do tipo “ok, recebido” antes de criar os próximos pedidos. Assim, a plataforma consegue criar milhares de pedidos por minuto. Se o sistema de atendimento ao

cliente cair, os pedidos continuam entrando e, quando o sistema voltar, ele "puxa" e processa todas as mensagens represadas no tópico, sem perda de informação.

A combinação de microsserviços e comunicação assíncrona possibilita desenvolver sistemas distribuídos ainda mais escaláveis, mais tolerantes a falhas, capazes de consumir e processar volumes colossais de dados em tempo real.

Mas só para lembrar! Embora a comunicação assíncrona seja mais eficiente para alto volume de dados, não é ideal para todo e qualquer cenário. A comunicação síncrona é útil para interações que exigem resposta imediata, como consultas a um banco de dados ou validação de dados em tempo real, a exemplo da autenticação. Por conta disso, é comum as plataformas oferecerem ambos os tipos de APIs.

Independente do tipo de comunicação, APIs são apenas interfaces, que assim como a interface visual, acessa algo que está "por trás", o que realmente executa as tarefas, o *back-end*.

2.3 Back-end

O back-end contempla tudo o que possibilita o funcionamento da tarefa: código, eventos, configurações, dados, infraestrutura. É a parte que fornece as funcionalidades, por exemplo, a capacidade de criar, adicionar itens e cancelar pedidos. As funcionalidades são desenvolvidas em linguagens de programação como Java, C#, Ruby, PHP, entre outras.

Esses componentes são as partes privadas da plataforma, portanto não precisam e não devem ser compartilhados com os consumidores da plataforma. Eles ficam protegidos pela interface.

2.4 Instância

O back-end e as APIs são hospedados na plataforma de tecnologia — como vimos no capítulo *O que é plataforma, afinal?* —, podendo ser um servidor próprio (*on-premise*) ou na nuvem, como AWS e Google Cloud. É onde se encontra a *instância*, máquina para onde se faz o deploy e onde o código é executado.

E se não bastassem tantas plataformas, ainda há as chamadas **plataformas de gerenciamento de API** (*API management platform*), que oferecem ferramentas que facilitam a gestão do ciclo de vida das APIs, como o Amazon API Gateway, da AWS; o Apigee, do Google Cloud; e os independentes MuleSoft, Kong e Gravitee.

API Gateway

O *API Gateway*, principal componente de uma plataforma de gerenciamento de API, intermedia a comunicação entre cliente e servidor. Ele recebe as requisições, aplica políticas de segurança, limite de uso (*throttling*) e transfere para o back-end. O back-end processa e devolve uma resposta para o gateway, que repassa ao sistema cliente.

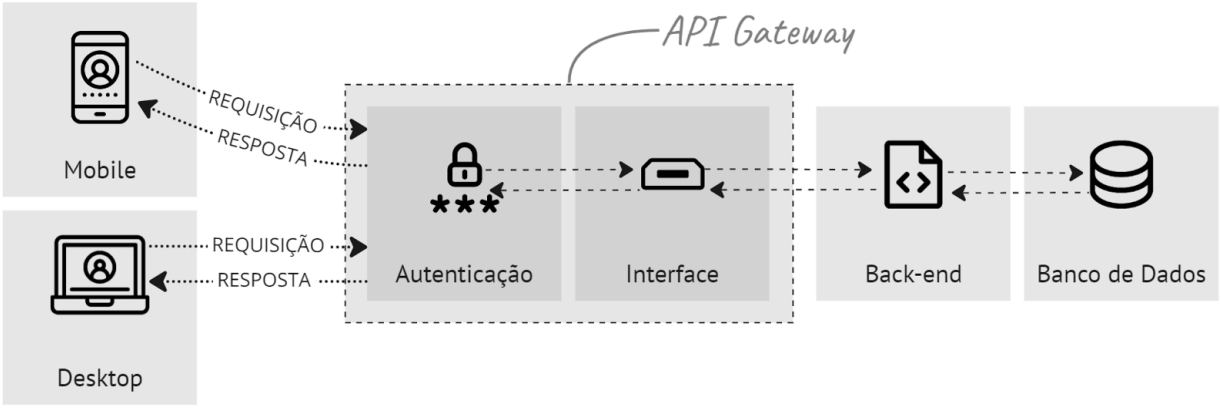


Figura 2.5: API Gateway.

O *gateway* também é responsável pelo balanceamento de carga (*load balance*), armazenamento em cache, controle de acesso

(autenticação e autorização), limitação de taxa (*rate limit*), coleta de dados e monitoramento. Todas essas funcionalidades, além de reduzir a carga no servidor de back-end, reduz a carga de trabalho para o time da plataforma — afinal, poupa o time de ter que criar esses recursos do zero.

Entretanto, como estamos falando de plataforma para uso interno, talvez não precise de *throttling* e *rate limit*, e o controle de acesso possa ser tão brando que não justifica uma ferramenta especializada no gerenciamento das APIs. Alguns recursos do servidor cloud e das plataformas de gerenciamento de APIs se sobrepõem. Por esses motivos, contratar mais um serviço talvez não compense.

2.5 Juntando tudo

A arquitetura de microsserviços, somada à comunicação via API, é o que habilita os recursos e funcionalidades a se tornarem blocos de montar (*building blocks*) e serem reutilizados indefinidamente, de maneiras diferentes. As novas composições também se tornam blocos reutilizáveis, permitindo a construção de produtos ainda mais complexos.

É comum as empresas começarem com APIs privadas e, de acordo com as oportunidades do mercado, abrir para o acesso público (com as devidas restrições, para não liberar ao público algo que coloque o negócio em risco). O contrário também pode acontecer. A empresa pode lançar uma API pública, que se torna mais importante para o desenvolvimento interno, e, no final das contas, o valor para o negócio ser maior para uso privado do que público.

Agora que sabemos as características da plataforma, vamos ver como é o ciclo de vida do desenvolvimento dela.

CAPÍTULO 3
Gestão do ciclo de vida

O **ciclo de vida do produto** refere-se à maturidade do produto no mercado, isto é, ao período entre a introdução de um produto no mercado até a retirada das prateleiras. É um conceito clássico do marketing usado para **tomada de decisões de negócios**, desde preços e promoções até expansão ou corte de custos, amparadas em como o mercado está reagindo ao produto.

Na prática, o ciclo de vida do produto está relacionado ao histórico de vendas. Como apontam os autores do livro *Gestão e desenvolvimento de produtos e marcas* (IRIGARAY et al., 2011): "*o ciclo de vida do produto é um modelo dos estágios do histórico de vendas e lucros de um produto*".

Não podemos confundir o *ciclo de vida do produto* com o *ciclo de vida do desenvolvimento do produto*. Obviamente, o desenvolvimento de um produto se dá antes de ele entrar no mercado, como mostra a imagem adiante.

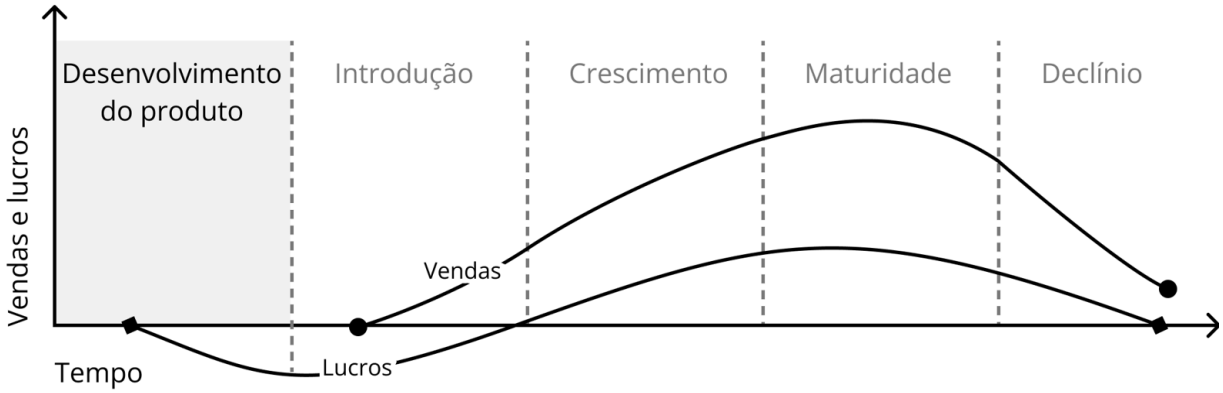


Figura 3.1: Curva do ciclo de vida de um produto. Baseado em: Gestão e desenvolvimento de produtos e marcas (IRIGARAY et al., 2011).

O **ciclo de vida de desenvolvimento do produto** refere-se ao processo de colocar um novo produto no mercado (ou melhorias de

um produto existente) e engloba desde o levantamento de ideias até o lançamento. É usado para **tomada de decisões acerca do produto** — o que será o produto; como ele será usado (design); como ele será construído (engenharia e tecnologia); como ele será disponibilizado no mercado; como será feita a sua manutenção —, com base nas necessidades e feedbacks dos clientes e usuários.

Nas quatro fases do ciclo de vida do produto (*introdução, crescimento, maturidade e declínio*), não há pontos de decisão sobre o desenvolvimento. É diferente do ciclo de vida de desenvolvimento, em que, a cada etapa, é avaliado se o produto continua no mercado ou se será abandonado (pivotado). Você já deve ter familiaridade com as etapas desse processo:

- **Discovery**: descoberta sobre o problema (necessidade ou oportunidade) e como resolvê-lo;
- **Delivery**: construção da solução;
- **Lançamento**: disponibilização da solução aos clientes;
- **Monitoramento**: mensuração e análise do desempenho da solução.

Quando o assunto é plataforma, encontramos algumas diferenças, a depender da fonte, mas é interessante notar que todas elas usam o termo **gestão do ciclo de vida da *API***, e não da *plataforma*. Talvez isso se deva ao fato de que a parte comercializável da plataforma são as APIs — entenderemos isso melhor nos próximos capítulos.

Para a Red Hat, líder mundial em software de nuvem *open source*, "*o ciclo de vida da API é duplo. O ciclo à esquerda envolve as atividades do produtor, enquanto o ciclo à direita envolve as atividades do consumidor*" (MASSÉ, 2019; tradução livre), como mostra a imagem a seguir, em que "produtor" produz APIs e "consumidor" as consome.

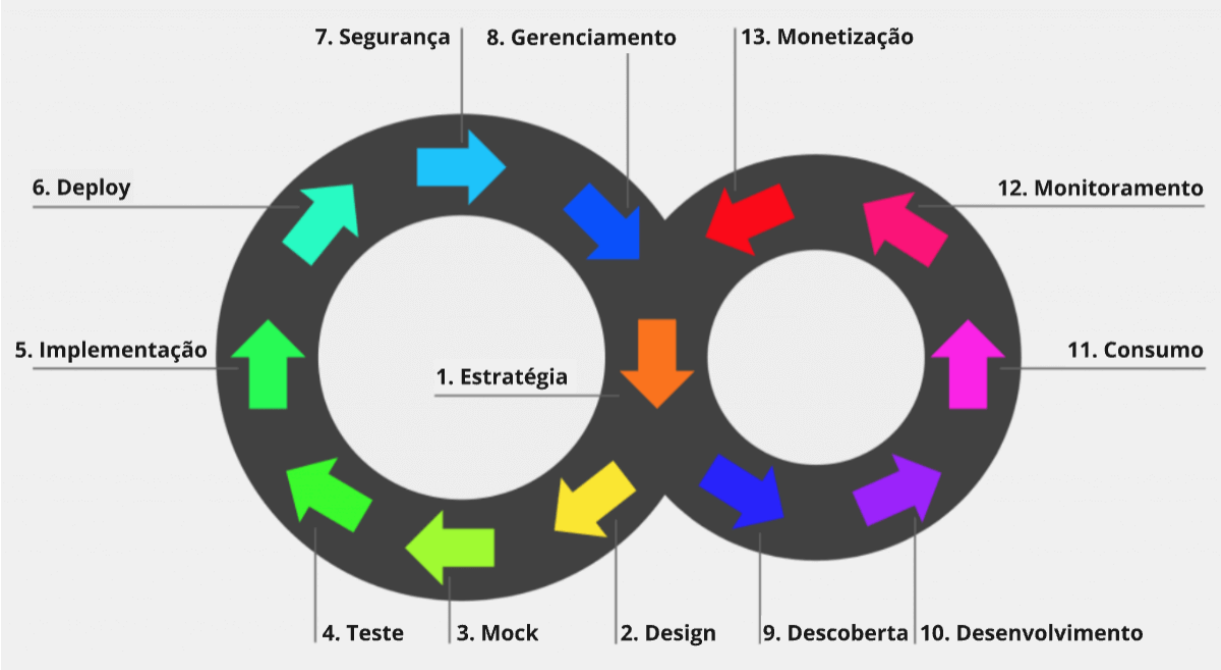


Figura 3.2: Ciclo de vida de API. Imagem adaptada de: Full API lifecycle management: A primer (MASSÉ, 2019).

Essa é apenas uma forma de expor o processo. No universo de produto, o lado esquerdo equivale ao ciclo de vida de desenvolvimento, enquanto o lado direito equivale à jornada do usuário (de API, no caso). As fases que a imagem mostra se encaixam como atividades dentro das quatro grandes etapas que conhecemos: *discovery*, *delivery*, lançamento e monitoramento.

3.1 Discovery

Na etapa de *discovery*, procuramos entender os problemas pelos quais os clientes passam e que também sejam oportunidades de

trazer resultados ao negócio. A partir disso, definimos quais oportunidades serão priorizadas.

Em busca de identificar a melhor aposta do momento e a solução ideal, temos um acervo vasto de técnicas que podemos utilizar: pesquisas, entrevistas, testes de usabilidade e de demanda do mercado, mapeamento da jornada do usuário, árvore de oportunidades, testes A/B, *story mapping*, *jobs-to-be-done*, *brainstorm*, *benchmark*, entre outras. A maioria delas também se aplica à plataforma e a APIs!

Design, Mock e Teste

Igualmente aos produtos *customer-facing*, nessa etapa descobrem-se hipóteses de soluções viáveis para o problema identificado e define-se o **design da API** —, aparência e comportamentos, entre outros elementos:

- Nomenclatura dos recursos;
- Propriedades de requisição e resposta;
- Especificações;
- Protocolos de segurança;
- Casos de uso;
- Acesso dos usuários.

Com os elementos definidos, já é possível criar o protótipo que será levado para o teste de usabilidade com os usuários das APIs, com o objetivo de validar a ideia. No universo das APIs, este protótipo é chamado de ***mock***.

Diferente da fase de **teste** do *discovery* de produto, que envolve o teste de usabilidade com os usuários e de mercado, a Red Hat atribui a essa fase o *teste de aceitação*, que detecta falhas e bugs, levando em conta o comportamento da API "mockada" (antes da implementação).

Uma vez que se chega à solução que melhor endereça os problemas, vamos para a próxima etapa, *delivery*.

3.2 Delivery

Implementação, Deploy e Segurança

A **implementação** dá vida à interface visual de produtos *customer-facing*, assim como à interface de plataformas, a API. Na etapa de desenvolvimento, guiado pelas especificações estabelecidas na fase de design da API, o time materializa o que foi idealizado e testa para garantir que tudo está funcionando conforme o esperado.

Feito isso, envia-se tudo o que foi construído para o ambiente de produção, o **deploy**. É provável que a empresa onde você trabalha já tenha adotado DevOps e utilize ferramentas de Integração Contínua/Implantação Contínua (*Continuous Integration/Continuous Delivery — CI/CD*) que automatiza o deploy. Se este for o caso, os testes de integração são executados como parte dessa *pipeline*.

A Red Hat adiciona a fase no ciclo de vida da API, **segurança**, como "um lembrete de que é um passo crucial" (MASSÉ, 2019; tradução livre). Isso não quer dizer que é um evento isolado depois do deploy. Ao contrário, quer dizer que as práticas de segurança devem ser incorporadas desde o início do desenvolvimento. Há diferentes formas de proteção aplicáveis ao longo do desenvolvimento até após o deploy, alguns exemplos são revisão de código (*code review*), teste de vulnerabilidade e autenticação dos usuários da API.

Gerenciamento

A fase de **gerenciamento** engloba versionamento, depreciação e descontinuação (*versioning, deprecation* e *retirement*). Versionamento está completamente relacionada à evolução, portanto é parte do ciclo de desenvolvimento do produto. Já depreciação e descontinuação têm mais relação com o estágio de declínio do ciclo de vida do produto, não do desenvolvimento. No entanto, são necessárias alterações no código e infraestrutura para "limpar" o legado, de modo que cabe no estágio de *delivery* do produto.

Pode-se retirar do mercado uma versão inteira de uma plataforma, retirar apenas algumas funcionalidades dela ou interromper a manutenção e o suporte a alguma versão, apesar de ainda permitir o uso. Isso pode ocorrer por vários motivos, como mudanças de tecnologia e questões de segurança.

Quando uma nova versão da API é construída, a versão anterior torna-se obsoleta e não recebe mais suporte e correção de bugs. Só porque uma API está obsoleta, não significa que ela não funcione mais. Versões anteriores de APIs podem permanecer em uso por muitos anos e ainda podem ser acessadas pelas aplicações.

Contudo, manter diferentes versões consome uma quantidade desproporcional de tempo e gestão. Exige ajustes que demandam a atenção do time da plataforma, atenção essa que pode ser melhor aplicada mantendo o foco na versão mais recente, principalmente no caso de plataformas internas, que não possuem milhares de aplicações conectadas ao redor do mundo que justifique esse trabalho extra.

Ainda assim, o planejamento para descontinuação é crítico e envolve todos os times que usam as *features* da plataforma nos apps deles. Esses times estão com foco nos objetivos que eles se comprometeram como parte da estratégia do negócio e não vão querer migrar só porque foi lançada uma nova versão. É uma das atividades que mais exige gerenciamento.

3.3 Lançamento

Uma vez que o deploy é finalizado, significa que está disponível para clientes acessarem e usarem. No contexto da plataforma, isso quer dizer que as APIs estão publicadas, prontas para serem integradas aos apps clientes. Sob a perspectiva de produto é quando inicia o ciclo de vida do produto, introdução. Já para a Red Hat (e alguns

outros players), o lado consumidor, é parte do ciclo de vida da API. Na prática, trata-se da jornada do cliente para ambos.

Descoberta, Desenvolvimento e Consumo

Imagine-se como cliente no começo de numa jornada. Como você compra algo que não sabe que existe? Não compra! Primeiro, você precisa saber que a coisa existe, depois, para o que serve e, se houver interesse, você vai querer saber como usa, o preço etc. O mesmo acontece com os clientes da plataforma. A Red Hat chama essa fase de **descoberta** (*discover*), mas em produto e marketing, é mais comum o uso do termo *awareness* (conhecimento). Para que potenciais clientes descubram ou conheçam as APIs, é necessário um **canal** onde eles consigam encontrar o conteúdo sobre as funcionalidades da plataforma e a documentação de como usá-las.

Se os clientes decidirem consumir a API, eles entram na fase de **desenvolvimento** para incorporar funcionalidades da plataforma no app deles. A partir do momento em que tudo está funcional, os apps clientes passam a consumir ativamente os dados da API, por isso a fase é nomeada de **consumo**.

3.4 Monitoramento

Após o lançamento do produto, é preciso saber se os resultados esperados foram alcançados e identificar áreas de melhoria ou novas necessidades. Além do feedback dos usuários, coletamos e analisamos métricas, como número de clientes e uso do produto, que orientam o planejamento das próximas evoluções do produto.

O que plataformas têm de diferente disso? Nada. Depois de lançar a primeira versão, a plataforma evolui e vem a próxima versão, a seguinte e por vai, fundamentada no feedback e nas métricas. Esta é a fase de **monitoramento**, tanto no ciclo de vida da API como no

de desenvolvimento de produto. Um ponto de atenção é que gama de métricas de plataformas e APIs é mais ampla. Além das métricas correlatas às de produto, monitoram-se também aspectos técnicos de engenharia e infraestrutura.

Para plataformas internas, o número de usuários das APIs talvez seja irrelevante. Provavelmente faz mais sentido saber se os apps clientes estão, de fato, usando as funcionalidades da plataforma. Se não estão, qual o motivo? Este pode ser um indicativo de que a plataforma precisa de melhorias ou flexibilidade para customização.

A grande diferença dessa fase é no dia a dia. PMs de apps geralmente só ficam sabendo de anormalidades quando chega alguma reclamação. PMs de plataforma, junto do time, devem monitorar a disponibilidade e a performance continuamente para detectar anormalidades antes que a operação seja afetada (ou de receber reclamação dos clientes) e, assim, agir preventivamente, por exemplo escalando infraestrutura.

3.5 Monetização

Dependendo do que a plataforma faz, pode chegar um momento em que vale a pena abri-la para consumidores externos e, quem sabe, gerar receita por meio de algum modelo de precificação. Se isso acontecer, a plataforma entra na fase de **monetização**.

Mas pensa um pouco. Plataformas internas não visam monetização direta. O retorno financeiro não reside na plataforma em si, mas no que ela representa em termos de potencial de gerar receita, habilitando a criação de novos negócios e participação no mercado. A plataforma contribui na rentabilidade da empresa quando os apps integrados a ela geram receita. Podemos dizer, então, que a monetização é indireta, portanto, é a forma de monetização da plataforma?

3.6 Plataforma para gerenciamento do ciclo de vida de APIs

Lembra da plataforma de gerenciamento de APIs que vimos na sessão *Instância* do capítulo anterior? Lá, o destaque foi o *API gateway*, mas esse tipo de plataforma — vou chamar de *gerenciador de APIs* daqui em diante para evitar confusão com a plataforma de produto — oferece mais do que um gateway. Oferece um conjunto de ferramentas para gerenciar o ciclo de vida da API desde a concepção até a retirada do mercado.

Na *implementação*, os gerenciadores de APIs facilitam o design, a criação documentação, testes e deploy. No *gerenciamento*, o controle de versões permite que as APIs evoluam sem quebrar a integração com aplicações consumidoras.

Na *descoberta* e *consumo*, o portal do desenvolvedor (*developer portal* ou *dev portal*) é o principal canal para consumidores das APIs, onde podem explorar as funcionalidades, acessar a documentação, obter credenciais de acesso e testar a API.

No *monitoramento*, dashboards com performance das APIs, como volume de tráfego, tempo de resposta e taxas de erro trazem visibilidade sobre o uso e a saúde da plataforma.

Na *monetização*, é possível criar planos com diferentes formas de precificação, sistema de cobranças e área para que os consumidores das APIs possam acompanhar o uso.

3.7 Juntando tudo

O time da plataforma desenvolve a solução, testa e faz deploy para o servidor. O servidor pode ser parte de uma plataforma de gerenciamento de API que cuidará da governança, gerenciamento do ciclo de vida, configuração de segurança, criação dos planos de pagamento. O time da plataforma cria o conteúdo para o portal do dev e depois disso, está pronta para consumo.

Quando consumidores querem adicionar funcionalidades da API no app que estão desenvolvendo, acessam o portal do desenvolvedor, se cadastram e obtém as credenciais. No começo do desenvolvimento adicionam as chaves das credenciais no código do app deles. O app faz a requisição à API e o fluxo passa pelo API gateway, que verifica as credenciais e autoriza (ou não) a troca de informações. A plataforma processa o que foi requisitado e devolve a resposta para o app cliente, onde os dados são exibidos ao cliente final.

Exceto quando surge alguma demanda urgente, o processo de criação e desenvolvimento de plataforma e APIs não inicia do nada. A estratégia — o primeiro item do ciclo de vida da API — e os objetivos *costumam* ser definidos previamente, assunto que veremos na Parte 2 do livro. Vamos nos aprofundar no ciclo de vida de desenvolvimento nas Partes 3 e 4, onde serão apresentadas as práticas de *discovery* e um apanhado do que vem depois disso.

Para fechar esta parte do livro sobre os fundamentos, vamos para o próximo capítulo conferir alguns dos desafios ao começar em um novo time e os papéis-chaves que vão ajudar você na gestão do ciclo de vida de desenvolvimento da plataforma.

Capítulo 4
Começando em um time ou plataforma

É provável que na empresa onde você trabalha o desenvolvimento da plataforma tenha iniciado organicamente, como uma iniciativa de pessoas de engenharia. Conforme as coisas ganham corpo, algum stakeholder diz precisar de algo e o time o desenvolve por ordem de chegada ou criticidade, no modelo de *projeto* de TI. Às vezes é pior, simplesmente se interrompe o que está em andamento, por conta da nova demanda que alguém considera crítica e, se bobear, nem necessária é.

Assim vai até que, em um dado momento, as coisas parecem caóticas. Muitas dependências, demandas de todos os lados, algumas aleatórias (outras não), desenvolvimento de coisas que não resolvem os problemas, falta de visibilidade e previsibilidade das entregas. Então, alguém decide que é necessária uma pessoa com olhar de produto para fazer a gestão.

É quando você, PM, recebe o convite para fazer a gestão da plataforma e, claro, a expectativa é de que você a trate como um produto. E o que isso significa? Significa que se deve considerar os usuários da plataforma antes de definir soluções e que também gerem retorno ao negócio.

É uma mudança sobre o que se entende por "sucesso". Enquanto em *gestão de projeto* o sucesso ocorre quando as partes são entregues (mesmo que gere pouco ou nenhum valor), em *gestão de produto* o sucesso é medido por resultados, por exemplo, aquisição de usuários, monetização, entre outros.

4.1 Mudança de projeto para produto

Sabemos que mudar o modo de trabalho não é simples assim.

Muitas vezes, o que se espera de uma pessoa PM é que ela receba as demandas (de preferência sem questionar muito), as priorize com base em algum framework e dê visibilidade sobre o andamento das coisas. Dessa forma, os stakeholders acreditam na suposta previsibilidade (e se apegam a ela) de quando as coisas que demandaram ficarão prontas. No fim das contas, o que eles querem é gestão de projetos, não de produto.

Isso não é exatamente um problema. O problema é acreditar que a demanda, que chega em forma de solução, é o que vai trazer o resultado esperado. Se esse é o caso, seja lá quem for que decidiu que era necessário uma pessoa com olhar de produto desconsiderou o problema, colocou foco na solução, do tipo "PM saberá priorizar e dar previsão de entrega". Solução errada! Project managers têm mais ferramental para cumprirem essa tarefa melhor do que product managers.

Não estou dizendo que gestão de produto não tem nada de projetos. Se o *discovery* traz respostas suficientes, o dia a dia do *delivery* é mais projeto e *status report*. Por essa razão, com frequência, há uma pessoa nas squads que olha para o fluxo de trabalho: o scrum master ou agilista. Quando não há, esta é uma das atribuições da pessoa PM.

Desmistificando ainda mais, no desenvolvimento de plataformas, tem muitos momentos em que as coisas se parecem mais com projetos, porque envolvem implementações estruturais para o funcionamento. É o caso de integrações e configurações de infraestrutura que, normalmente, o time sabe o que e como deve ser feito — exceto quando a tecnologia é nova para o time; nesse caso, exige *discovery* técnico e PoCs (*Proof of Concept* — prova de conceito).

É como a construção de um prédio. No que dependesse das pessoas de negócio, a primeira coisa a ser construída seria a que trouxesse o

maior resultado para a construtora, o apartamento da cobertura. Mas, para a infelicidade delas, é necessário começar pela fundação que, apesar de ninguém ver, é o que torna possível construir cada andar. É ela que tem a capacidade de sustentar a somatória de todos os andares.

Da mesma forma são as plataformas. Elas liberam capacidades, *features*, o que pode deixar você desconfortável depois de tanto ouvir ou ler o Marty Cagan (2019) dizer que *product teams* atuam completamente diferente de *feature teams* e que o foco de verdadeiros PMs é *outcomes*, não *outputs*. Então, para desconstruir essa mentalidade, quero lembrar que só é possível obter *outcome* se houver *output*.

Voltando ao exemplo do prédio, as torneiras do apartamento, quando abertas, liberam água e, ao fechá-las, a água para (*features*). Elas fornecem a capacidade de um banho quentinho (*outcome*), mas, para isso, é necessária a instalação do encanamento, das torneiras e a conexão com o sistema hidráulico da cidade (*outputs*).

Como plataformas são a base para desenvolver produtos, com frequência vamos trabalhar na fundação e em *features* que se tornarão *outcomes* se, e somente se, as *features* forem adicionadas aos produtos da empresa. Portanto, no pior dos cenários, há entrega de *feature* e ela não é adicionada aos apps. É para evitar esse tipo de acontecimento que exercemos nosso papel de PM.

Devemos entender a necessidade dos times de apps e o impacto da funcionalidade para o negócio. Se ela for realmente importante, dificilmente os times vão desistir dela e, consequentemente, a plataforma entra na cadeia de valor.

Então, eu diria que podemos até lidar com coisas que são aparentemente projetos, mas que entram pelo motivo certo e no momento certo. Aliás, qualquer produto que começa do zero, até entregar valor, tem a construção estrutural. A diferença é como

vamos escolher desenvolvê-la, seguindo o modelo *waterfall* ou iterativo/incremental de alguma metodologia ágil.

Podemos fazer essa mudança de ponto de vista em nós mesmos, mas a empresa também precisa mudar o entendimento que tem sobre o que é sucesso para plataformas — no capítulo *Métricas*, entraremos mais neste assunto.

4.2 Time quase multidisciplinar

Outra coisa com que precisamos nos acostumar é com a formação do time. Sabemos que é comum os times de produto serem multidisciplinares, como descrevem os autores Marty Cagan e Chris Jones no livro *Empowered*:

"*No modelo de time de produto empoderado, o product manager tem responsabilidade clara, garantir que as soluções sejam* valiosas *(nossos clientes comprarão o produto e/ou escolherão usá-lo) e* viáveis *(atenderão às necessidades do negócio). Junto a um product designer, responsável por garantir que a solução seja usável, e um tech lead, responsável por garantir que a solução seja factível, o time é capaz de colaborar para endereçar toda a gama de riscos (valor, viabilidade, usabilidade e factibilidade). Juntos, assumem o problema e são responsáveis pelos resultados.*" (CAGAN; JONES, 2011; tradução livre)

Times de plataforma não costumam ser multidisciplinares. Dificilmente terá uma pessoa product designer — a não ser que a plataforma precise de interface visual, algo no estilo *low-code* ou *no-code*, como o Twilio Studio exibido na próxima adiante.

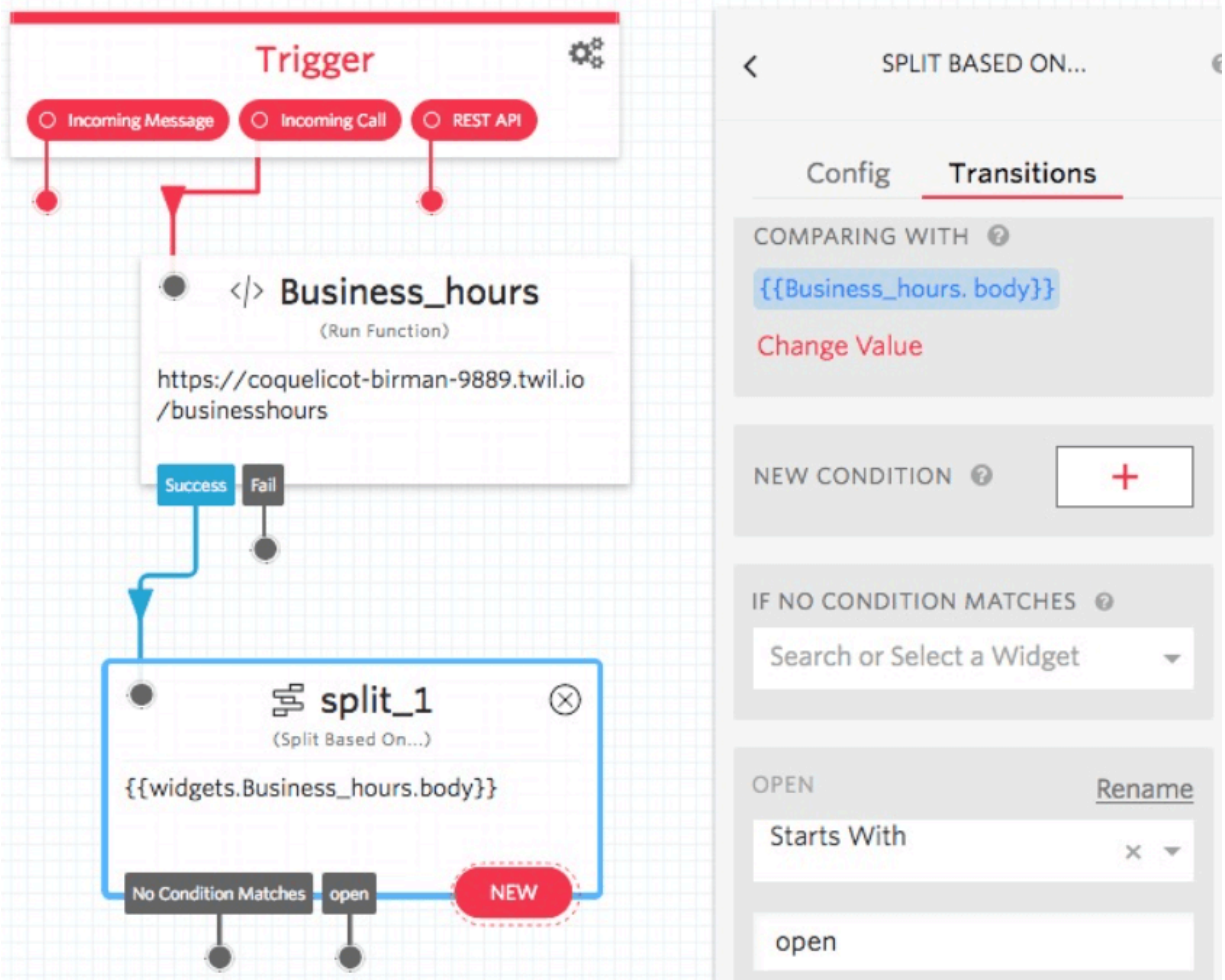


Figura 4.1: Twilio Studio. Fonte: Twilio Blog (THURSTON, 2019).

Sabemos o nosso papel como PM e já estamos familiarizados com o papel do tech lead, mas plataformas são construídas a partir do design das APIs e da arquitetura. Então, quem fica responsável por essa tarefa? A pessoa API designer.

API designer

Semelhante ao product designer dos times de produto, o API designer é a pessoa responsável pela experiência do usuário e do "*look and feel*" da interface. Quando falamos de interface de app (UI — *User Interface*), sabemos do que se trata, porque conseguimos ver, experienciar. Agora, que cara tem a interface de uma API? Tem

cara de texto, código, como veremos em detalhes parte 3, *Discovery de plataforma e API*.

Quando entrei para o time da Order Platform, havia 18 pessoas, todas de engenharia, back-end; entre elas, 5 especialistas (equivalente a managers na carreira em Y) e 3 engineering managers. Não tinha uma pessoa designada para o papel de API designer. Eu, intuitivamente, o atribuiria ao tech lead, mas os cinco especialistas eram a referência técnica para o time.

Eu já tinha estudado o assunto e, depois de mais de um mês no time — só depois de perceber os tipos de tarefas, cerimônias e rotinas —, perguntei quem poderia fazer o papel de API designer. Ouvi aquele silêncio desconfortável. Imaginei que eu tinha feito uma pergunta incabível. O time desconhecia esse papel, então preferi explicar em que essa pessoa atuaria e a resposta foi: qualquer pessoa do time tem capacidade de fazer isso.

Em tese, todos os devs têm conhecimentos para esse papel, por se tratar de um assunto corriqueiro no dia a dia do desenvolvimento. Mas só em tese. A verdade é que o API designer toma decisões técnicas que vão além do design da interface; ele toma decisões em torno de desafios obscuros e complexos de ciência da computação, como arquitetura que garanta escalabilidade, resiliência, extensibilidade, reutilização, segurança e performance, para citar alguns.

Além das competências técnicas, a pessoa deve ter visão ampla para conciliar necessidades do negócio, dos usuários da plataforma e da TI, e isso exige experiência, mais senioridade.

A lista de atribuições é vasta. Sintetizando o que se encontra em alguns livros e descrição de vagas, as habilidades do API designer se concentram em:

- Conhecer e aplicar melhores práticas de design de API e arquitetura;

- Definir padrões de design, como nomenclaturas e implementação de segurança das APIs — autenticação, autorização, controle de versão, coleta e análise de dados para gerenciamento de tráfego e performance;
- Compreender a visão do negócio e traduzi-lo em funcionalidades e design de API consistente, fácil de entender e usar;
- Prototipar o design da API e testar;
- Orientar o time na criação, desenvolvimento e testes das APIs;
- Contribuir na criação e melhorias da documentação, tutoriais, bibliotecas e código de exemplo, visando à experiência do dev usuário;
- Entender o dev usuário, facilitando as sessões de feedback como parte do processo de design da API;
- Interagir com stakeholders para entender novos casos de uso e melhorias necessárias nas APIs existentes;
- Antecipar necessidades futuras e definir o design com reutilização em mente;
- Monitorar a plataforma e contribuir na solução de incidentes.

Você percebe que, com essas atribuições, o API designer se torna a referência técnica do time? Então, eu diria que a pessoa tech lead pode exercer esse papel. Assim sendo, além de a tech lead nos ajudar a entender se algo é factível, ela também ajudará a torná-lo usável e a melhorar a experiência do dev usuário. Em outras palavras, essa pessoa pode ser tech lead e API designer.

Como dito, havia cinco pessoas no time da Order Platform que poderiam exercer o papel de API designer. No entanto, algumas pessoas não gostam de reuniões, de entrevistar usuários. Então, ficou livre: aquela que quisesse participar das conversas acerca de determinado tema trazia o desafio técnico para o time e outro especialista poderia definir o design. Isso feito, fazíamos o equivalente ao *design critique*, em que juntávamos os demais especialistas e os engineering managers do time para coletar feedback e incrementar a solução.

Se na empresa em que você trabalha tem *software architect*, talvez seja essa a pessoa indicada para este papel. A propósito, no Google, não se chama API designer, mas API architect.

É importante saber que o API designer não tem a responsabilidade de se atentar aos artefatos visuais, então um product designer pode ser alocado temporariamente para ajudar a pensar na experiência do portal do desenvolvedor, por exemplo. Além de API designer ser algo novo para o time, algumas das pessoas podem já ter trabalhado com PMs antes, outras não, por isso você precisa alinhar com elas qual é o seu papel.

Evangelizar é preciso

Sem dúvida, há diferenças entre gestão de produto e gestão de plataforma, mas o nosso papel é o mesmo: garantir que cada recurso, cada funcionalidade, cada artefato tenha valor e seja viável.

Talvez algumas pessoas do time não saibam disso. Algumas delas confundem, pensam que é mais um(a) "chefe". Foi o que aconteceu no time da Order Plataform. Então, pensei: preciso evangelizar.

Preparei o material e apresentei. Mostrei o conceito de time multidisciplinar, o modelo Spotify de squad, o modo de trabalho entre o trio PM, product designer e tech lead, e a minha interpretação de como poderíamos replicar, separando os papéis e as responsabilidades. Deixei claro que eu não era chefe de ninguém e que não fazia gestão de pessoas, fazia gestão da plataforma e mostrei onde PM atua e quando atua.

	Lidera / Executa	Participa / Acompanha
Gestão das pessoas do time	Eng Manag	
Define o que fazer (features)	PM	Tech Lead, Dev, API Design
Define quando fazer (priorização)	PM	Tech Lead, Dev, API Design
Define como fazer (solução)	Dev, API Design	Tech Lead, PM
Define quem vai fazer	Tech Lead, Dev, API Design	
Gestão de roadmap	PM	Tech Lead, Dev, API Design
Gestão de backlog	PM, Tech Lead	Dev, API Design
Gestão de fluxo de trabalho	Eng Manag	Tech Lead, Dev, API Design, PM
Define a estratégia da empresa	Alta gestão	

Figura 4.2: Apresentação papéis e responsabilidades para o time.

Algumas pessoas tinham trabalhado com PMs, mas nenhuma delas tinha clareza, especialmente no contexto de plataforma. Inúmeras vezes precisei explicar o básico, como o que é esperado de determinada atividade de discovery, conceitos de agilidade, *dual-track*, experiência do usuário.

Se você for fazer algo desse tipo, é imprescindível que antes você tenha clareza sobre como você contribui e onde precisará de contribuição do time.

4.3 Os primeiros 90 dias

No livro *The First 90 Days*, Michael Watkins (2013) afirma que, quando fazemos transição de carreira ou de empresa, desde o início

consumimos valor da empresa e, à medida que aprendemos e passamos a agir, começamos a criar valor.

O ponto em que passamos a gerar mais valor do que consumimos é chamado de **ponto de equilíbrio** (*break-even*). Deste ponto em diante, espera-se que a pessoa contribua continuamente na geração de valor.

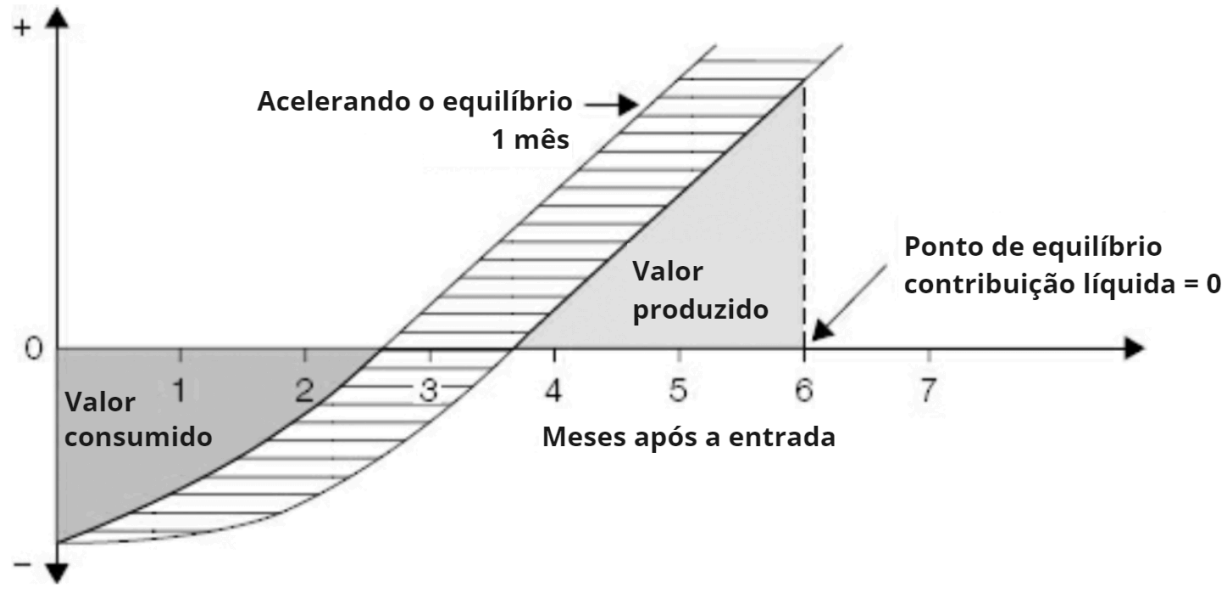


Figura 4.3: Ponto de equilíbrio. Imagem adaptada de: The First 90 Days (WATKINS, 2013).

Claro que isso depende do cenário, mas leva em torno de 90 dias para se atingir o ponto de equilíbrio, segundo o estudo do autor.

O seu conhecimento é muito valioso. Ele pode e deve ser usado para contribuir na evolução da plataforma e nas expectativas dos times dos apps — só tenha cuidado, porque você está numa nova empresa (ou plataforma) e, talvez, a maior armadilha seja supor que o que tornou você bem-sucedido(a) no trabalho anterior é o que manterá o seu sucesso.

Nas primeiras semanas, você tem que identificar oportunidades para construir sua credibilidade. Nos primeiros 90 dias, você precisa identificar maneiras de criar valor e melhorar os resultados do

negócio que ajudarão você a chegar ao ponto de equilíbrio mais rapidamente.

No começo, você receberá uma enxurrada de informações. É capaz que algum stakeholder empolgado, sabendo que agora tem PM no time, chegue com demandas na primeira semana. O excesso de informações vai tirar o foco do que você precisa aprender agora, o que vai aumentar o tempo para alcançar o ponto de equilíbrio. Quanto mais rápido você escalar a curva, mais cedo poderá começar a contribuir e tomar boas decisões.

Cuidado para não se tornar vítima da pressa em agir, ou como dizem, "mostrar serviço". Quando isso acontece, você direciona o seu esforço em agir cedo demais e fica "sem tempo" para aprender. Você pode tomar decisões erradas ou ruins e, como consequência, encontrar resistência às suas iniciativas e arruinar a sua credibilidade.

Tudo terá o seu devido tempo. Você tem 90 dias para aprender e, então, começar a gerar resultados significativos. O mais importante é usar esse tempo com sabedoria, fazer as escolhas certas sobre **o que aprender** nessa fase.

Então, aqui vai uma ideia de plano para ajudar você a se concentrar no que aprender nessa fase.

Primeiro mês

Ao final do primeiro mês, você deve ter conhecido o suficiente sobre a empresa e sobre a plataforma. Para chegar lá, conhecer o negócio vai te dar visão sobre o contexto em que a plataforma está inserida e como se encaixa no ecossistema.

Estas são perguntas **sobre a empresa** para as quais você pode procurar respostas:

- A qual segmento de mercado a empresa atende?
- Quais são os produtos e o público-alvo de cada um deles?

- Como a empresa está organizada? Quais unidades de negócios a empresa tem e a quais delas a plataforma atende?

Saber sobre o segmento de mercado ajuda a entender alguns requisitos. Por exemplo, o segmento de pagamentos é fortemente regulamentado. Se a plataforma está relacionada ao processamento de pagamentos, ela deve seguir as regulamentações à risca. Dependendo do tamanho da organização, esse aprendizado pode levar meses. Por enquanto, você precisa de uma visão geral para se situar. Provavelmente, nas primeiras semanas você vai conhecer a liderança de produto, de engenharia e, obviamente, o time. Nesses encontros você deve buscar aprender **sobre a plataforma**.

A primeira recomendação é relembrar às pessoas com as quais você conversar que gestão de plataforma é algo novo para você e que você está aprendendo. Dessa forma, as pessoas ficarão mais dispostas a ajudar ao invés de cobrar. A última coisa que você deveria sofrer são cobranças neste momento e, dependendo do quão novo isso é para você e da complexidade da plataforma, a expressão "estou aprendendo" acompanhará você por mais do que 90 dias, infelizmente.

Por exemplo, a Order Platform é densa e complexa. Há pessoas que estão há mais de 3 anos lá e não conhecem a plataforma toda. Agora, imagine uma pessoa PM, sem background em engenharia, nesse contexto. O aprendizado é diário. Então, no começo, procure conhecer a plataforma no nível macro e, ao longo do tempo, vá aprofundando o aprendizado de um tema por vez.

Algumas respostas que você vai precisar para conseguir para fazer a gestão da plataforma:

- Em qual parte do ecossistema a plataforma está inserida?
- O que a plataforma faz? Quais são as funcionalidades? Como essas funcionalidades aparecem nos produtos *customer-facing*? O que o usuário final consegue fazer com essas funcionalidades?

- Quais times utilizam as funcionalidades? Como usam? Por que usam?
- Quais problemas ou fragilidades a plataforma apresenta hoje?
- O que já tem de *discovery*?

Essa é uma boa oportunidade para documentar no *Business Model Canvas* ou no *Lean Canvas*, ou na ferramenta que você prefere ou considera mais adequada para o estágio do ciclo de vida em que a plataforma está — na Parte 2 do livro, vamos explorar o Lean Canvas.

Olhar para o passado nos ajuda a entender o presente e planejar o futuro. Então, desse aprendizado você terá um diagnóstico que também guiará onde atuar. Para isso, você precisa saber, por exemplo:

- Como o time está estruturado? Quem são as pessoas do time e qual papel cada uma exerce?
- Quais são os processos e rituais do time? Como estão esses processos? São bem estabelecidos? São controlados, metrificados?
- O que o time já fez que deu certo? O que deu errado e por quê?
- Qual a visão da plataforma? Foi definida alguma estratégia? Essa estratégia está alinhada ao negócio? O time está, de fato, seguindo-a? Se não, por que não? Se sim, a estratégia levará a organização aonde ela precisa ir?
- Os objetivos e as entregas são precisos? Tem coisas atrasadas?

Em linhas gerais, com essas informações, você vai entender por que as coisas são feitas do jeito que são, se essas razões ainda são válidas, se existem planos de mudanças e, principalmente, se tem algum ponto que você consegue colaborar. É importante procurar pontos pequenos, que já estão em andamento, para colaborar. Tentar fazer demais vai tirar o foco do aprendizado que você precisa neste momento. Levará você a atirar em várias direções, iniciar muitas coisas, o que deixará as pessoas do time confusas, sem entender o que é realmente importante.

Como mencionado anteriormente, é comum iniciativas de "plataformização" começarem sem PM. Se esse for o caso, entre o primeiro e o segundo mês, você precisará evangelizar.

Segundo mês

Se o aprendizado do primeiro mês ocorrer como o esperado, no segundo você será capaz de explorar as hipóteses e expectativas.

Primeiro, tente conversar com pessoas da média e alta gestão para saber quais as expectativas acerca da plataforma. Procure entender também qual a **visão de futuro** para a empresa, porque, ao contrário dos produtos *customer-facing* que visam ações de curto-médio prazo, plataformas sustentam o crescimento da empresa no longo prazo.

Isso feito, é hora de ouvir as expectativas das pessoas dos times que usam a plataforma. Em um plano geral, como a plataforma contribui para o produto delas? Que problemas enfrentam por causa da plataforma? O que está ruim para o usuário final? Quais os planos para o futuro dos produtos delas?

Com base nos insumos que você coletou do time, da média e alta gestão e dos usuários da plataforma, você terá percepções semelhantes e diferentes sobre os problemas e oportunidades.

Lembre-se de documentar e atualizar o Canvas ao longo do processo e de, entre o segundo e o terceiro mês, chamar o time para compartilhar as informações e entender os pontos de maior risco. Essa é a hora em que você (junto com o time) conseguirá definir por onde começar o *discovery*.

Terceiro mês

Depois de entender o que tem em aberto para se descobrir, é hora de selecionar o "discovery da vez" e chamar os times que usam a plataforma para entrevistas. Após isso, você terá clareza sobre o que

deve ser feito e o porquê, para, junto do time, definir quando será feito e o que é necessário para fazê-lo.

Expondo dessa forma, parece simples e linear, mas não é! Ao longo desse caminho, a plataforma passará por incidentes, pessoas vão chamar você para passar demandas que ninguém esperava, haverá reuniões com os times dos app (e outras plataformas) para negociar e ajudar a viabilizar o desenvolvimento do app deles, eventos e ritos da empresa, e por aí vai.

Além disso, os times com os quais você precisa falar também têm os produtos deles para evoluir, e isso significa que não estão dedicados a ajudar na evolução do seu, o que pode tornar o processo mais demorado.

Talvez no terceiro mês você ainda não tenha certeza do que deve ser feito, mas terá muito mais conhecimento do que quando começou. A pessoa PM que você é no terceiro mês não é mais aquela do primeiro. Isso já é um baita progresso!

4.4 Juntando tudo

Desenvolver plataforma é mais do que um desafio técnico. Elas existem para resolver problemas de negócio. As tecnicidades são apenas uma parte da história, parte que já tem o grande elenco das pessoas de engenharia: developers, tech leads e engineering managers. Será que mais uma pessoa com background técnico é o que faz a diferença quando o assunto é a gestão da plataforma como produto?

A não ser que a empresa espera que a pessoa PM tome as decisões técnicas, como de arquitetura, minha experiência mostra que não. O

que a empresa e o time mais precisam é de alguém para pôr ordem na casa e que vise à entrega de valor.

Meu par nessa empreitada, Danilo Souza, engineering manager da Order Platform há mais de 3 anos, disse:

"*Desde a criação do time até a plataforma ser encarada como produto, as metas não refletiam as necessidades do negócio, estavam lá por inércia. Os times de produtos* customer-facing *definiam suas metas de acordo com a estratégia da empresa, e os produtos que dependiam da Order Platform para evoluir obrigavam o nosso time a mudar as metas — e é claro que o planejamento não passava do primeiro mês sem alterações. Logo surgiam metas de outros times desavisados que deveriam ser priorizadas no planejamento da nossa plataforma.*"

O que mudou? De um time que tinha a percepção de que a plataforma causava baixo impacto e que tinha pouca visibilidade para o negócio, mudou para um time alinhado aos objetivos, sabendo o que fazer no ciclo, com o valor da plataforma aderente à estratégia do negócio e com o apoio da alta gestão.

Como fizemos isso? Bom, este é o assunto do livro inteiro. Nem haveria aprendizados para compartilhar aqui se eu não tivesse conseguido atuar como PM da plataforma. Mas fazê-lo seria impossível sem o apoio dos engineering managers, que fizeram questão de abraçar a cultura de produto. Eles ajudaram a evangelizar e fazer as práticas de *discovery* darem certo, por exemplo. Firmar essa parceria é imprescindível para conseguir exercer o seu papel de PM.

E, para exercê-lo, recomendei anteriormente documentar o aprendizado usando Canvas. É o que vamos fazer na próxima parte do livro.

PARTE 2 — Estratégia para plataforma e API

CAPÍTULO 5
Em busca de platform-market fit

Só para relembrar, *product-market fit* é o ponto em que os resultados demonstram que um produto atende às necessidades de um mercado grande o suficiente para gerar retorno à empresa. Uma vez alcançado, o produto entra no próximo estágio, *growth*. Então, o foco deixa de ser *product-market fit* e passa a ser crescimento de base e, em alguns casos, rentabilid... Calma! O foco deixa de ser *product-market fit*!?

Vamos refletir. Por que empresas gigantes, como Kodak, Atari, Blackberry, Blockbuster, entre tantas outras, morreram? Os produtos delas deixaram de ter *fit* com as necessidades e desejos emergentes do mercado. Então, eu diria que **manter o *product-market fit* é tão importante quanto alcançá-lo**. É o que motiva a evolução contínua do produto, o que prolonga a vida útil, ou seja, adia entrada ao estágio de declínio do ciclo de vida.

Plataformas precisam de alguém que mantenha o seu *product-market fit*. Veja bem, não é sobre os apps usados pelos usuários finais, é sobre a plataforma — para não confundir com o *fit* de produtos *customer-facing*, vou chamar de ***platform-market fit***.

Reparou no termo ***manter*** o *platform-market fit*? Isso porque, teoricamente, o *fit* com o mercado já foi alcançado. Primeiro, o mercado é interno, portanto, conhecido. Segundo, independente do estágio em que a plataforma está, os recursos e as funcionalidades já existem em produtos da empresa (muitas vezes, é justamente o que motiva a plataformização). Portanto, até certo ponto, *o que* precisa ser feito também é conhecido; o *como* é o desafio técnico

que precisa ser descoberto e nós, PMs sem background técnico, não dominamos.

Plataformas também precisam ser planejadas e projetadas apropriadamente para alcançar e manter o *platform-market fit*. Devemos garantir que qualquer nova demanda esteja alinhada à visão e à entrega de valor para os diferentes apps plugados na plataforma e, claro, à experiência do usuário — entraremos nesse assunto nos próximos capítulos.

Só porque o mercado é interno, pode parecer que não precisa de esforço para que a plataforma tenha *fit* com o mercado. Então, pense: o que acontece se a plataforma não é adequada o suficiente ao que o cliente interno precisa? O cliente pode optar por usar uma versão *open source* ou desenvolver a própria solução do zero. Então, para que a plataforma existe se, no fim das contas, não atende bem e, por consequência, não traz o retorno que a empresa espera?

Quando isso acontece, algum stakeholder tem a ideia "genial" de obrigar os times a usarem a plataforma que foi desenvolvida internamente, e assim justificar o que foi investido até então. É quando o time da plataforma é bombardeado por demandas de todos os lados, a fim de fechar o *gap* entre o que a plataforma faz e o que ela deveria fazer para atender às necessidades dos clientes internos e suportar o negócio.

Esse é ou não é um problema de *platform-market fit*?

A plataforma pode ser de altíssima tecnologia, bem construída, seguir todas as melhores práticas existentes, não ter um bug sequer e, ainda assim, ser inútil às necessidades. Isso mostra que não é um problema tecnológico, não é algo a ser resolvido com mais engenharia. O que resolve é a boa gestão da plataforma, a começar pela estratégia.

5.1 Plataforma como estratégia da empresa

A empresa que ainda não alcançou o *product-market fit* lida com incertezas demais e precisa testar muito, pivotar, fazer o que for necessário para alcançá-lo rapidamente. Este não é um bom momento para começar a construir plataformas para sustentar o produto. A dependência da plataforma vai desacelerar o desenvolvimento do produto e retardar o *product-market fit*.

Uma vez que a empresa ganhou certa escala e oferta mais produtos no mercado, emergem problemas. É quando faz sentido aderir à estratégia de plataforma de produto, isto é, **plataformização é estratégia por si só** (ao menos deveria ser).

O sucesso da estratégia de plataforma depende do *buy-in* de toda a empresa, porque requer a orquestração de várias áreas e até mudanças no design organizacional. A execução leva vários ciclos, sem retorno imediato.

Por conta disso, a alta gestão precisa ser consciente e cuidadosa antes de aderir à estratégia de plataforma. É fundamental que a estratégia seja liderada pela pessoa CTO, apoiada pela CEO. A Amazon não teria condições de encarar esse tipo de transformação se o direcionamento não tivesse partido do Jeff Bezos, fundador da Amazon, para toda a organização. É isso mesmo: tem que ser *top-down*.

Mesmo em condições ideais — CTO como patrocinador (*sponsor*) e CEO comprado com a estratégia —, o investimento pode ser tímido até que a plataforma entregue valor. Claro que mesmo assim virão cobranças como "quando teremos a primeira versão da plataforma?", ou, "quando teremos o retorno sobre o investimento?". Então, os times de plataforma convivem com o dilema de equilibrar as expectativas de curto prazo e construir uma plataforma que vislumbra o futuro da empresa, um investimento de longo prazo. A depender do apetite da empresa, o patrocinador pode investir em mais recursos com o intuito de antecipar os prazos. Mais recursos

significam mais gastos e, de brinde, vem junto mais ansiedade pelo retorno, elevando ainda mais as expectativas de curto prazo.

Comparando a empresas nativas digitais e bem estabelecidas no mercado (Google, Uber, Amazon), estamos nos primeiros níveis de maturidade de plataformização. Talvez a empresa na qual você trabalha não tenha chegado ao ponto em que a plataforma é parte essencial da estratégia. Aliás, neste contexto, é comum plataformas serem vistas meramente como suporte ou, ainda pior, não haver uma divisão clara entre plataforma e produto (*customer-facing*).

No segundo caso, o mesmo time desenvolve a plataforma e parte do app que deriva dela. É como se a plataforma fosse apenas o back-end dos produtos que ela suporta. E, quando a plataforma serve a mais de um app, o time trabalha no front-end de cada um deles. Por conta disso, atende a públicos além dos usuários da plataforma.

Independente do contexto, essas coisas continuarão a acontecer até que as empresas adquiram certa maturidade no assunto.
Começando pela sua plataforma, é importante definir a estratégia — pelo menos você e o time saberão aonde querem chegar e conseguirão planejar melhor como chegar lá.

5.2 Definindo a estratégia da plataforma

Partindo do pressuposto de que a empresa já adotou ou está em vias de adotar a estratégia de plataforma, não importa se você vai atuar em uma plataforma existente ou nova, a primeira coisa que você deveria procurar saber é: por que a empresa acredita que essa plataforma deva existir?

Com base na resposta, será possível estabelecer o propósito da plataforma, identificar o que se pretende alcançar, o problema que vai resolver e a oportunidade que vai tentar aproveitar. Indo além, definir **quais soluções a plataforma tornará possível que os**

times de produto desenvolvam. Com isso, chega-se ao básico para definir a estratégia da sua plataforma.

Atenção para não confundir "plataforma como estratégia da empresa" com "estratégia da plataforma" (individual) que compõe a estratégia da empresa.

Figura 5.1: Plataforma como estratégia da empresa versus Estratégia de uma plataforma.

A empresa que adota a estratégia de plataforma visa a uma destas coisas (ou a todas): acelerar o *time-to-market*, diminuir os custos, diminuir os riscos. A estratégia de uma plataforma (individualmente) deve levar a empresa a alcançar esses objetivos. Ao definir a

estratégia, você deve conseguir responder à pergunta *"Esta plataforma visa a quê?"* — aqui não tem o que florear.

Vimos a definição sobre plataforma de produto com base na teoria e práticas clássicas — o caso Volkswagen abordado no capítulo *O que é plataforma, afinal?*. Você acredita que quase tudo o aprendemos sobre gestão de produtos é releitura de conceitos clássicos? Por isso, minha teoria é de que, independente do tipo de produto — xampu, carro, satélite e, obviamente, os digitais, inclusive plataformas —, os conceitos essenciais de gestão de produtos se aplicam a todos eles. Duvida? Veja, então, este clássico do marketing.

Pelo que se sabe, a história da gestão de produtos iniciou no departamento de marketing da Procter & Gamble, em 1931. Em princípio chamado de "gerente de marcas" (*brand man*) e depois de "gerente de produtos e marcas". Em linhas gerais, essa pessoa deveria cuidar de todos os aspectos da marca:

- Conhecer pessoalmente os clientes e distribuidores para descobrir problemas;
- Desenvolver o plano para resolver o problema;
- Desenvolver o plano para lançamento (*go-to-market*);
- Monitorar os resultados.

Mr. N. H. McElroy May 13, 1931

Mr. R. F. Rogan

ADV**N. H. MCELROY

Because I think it may be of some help to you in putting through our recommendation for additional men for the Promotion Department, I am outlining briefly below the duties and responsibilities of the brand men.

This outline does not represent the situation as it is but as we will have it when we have sufficient man power. In past years the brand men have been forced to do work that should have been passed on to assistant brand men, if they had been available and equal to the job.

Brand Man

(1) Study carefully shipments of his brands by units:

(2) Where brand development is heavy and where it is progressing, examine carefully the combination of effort that seems to be clicking and try to apply this same treatment to other territories that are comparable.

(3) Where brand development is light

(a) Study the past advertising and promotional history of the brand; study the territory personally at first hand - both dealers and consumers - in order to find out the trouble.

Figura 5.2: Memorando da Procter & Gamble. Fonte: Wikimedia Commons (https://en.wikipedia.org/w/index.php?title=File:Neil_Mcelroy%27s_1931_Brand_Man_Memo.pdf&page=3).

Ainda com base na administração de marketing (KOTLER; KELLER, 2018), o plano estratégico e tático tem como processo:

1. Segmentação do **mercado** de acordo com características demográficas, psicográficas e comportamentais;

2. Seleção do **público-alvo** que possui determinada **necessidade ou desejo** dentro do segmento que apresenta a maior oportunidade (demanda e lucratividade);
3. Definição da **oferta de mercado** (produtos e/ou serviços que tangibilizam a **proposta de valor** que vai satisfazer a necessidade ou desejo) e do **posicionamento no mercado**;
4. Definição das características específicas do **produto**, o **preço** e a **distribuição**;
5. Seleção dos **canais** para promoção: internet, propaganda, força de vendas e outros meios de comunicação.

Não é exatamente isso o que compõe a ferramenta consagrada dos cursos gestão de produtos digitais, o *Business Model Canvas*?

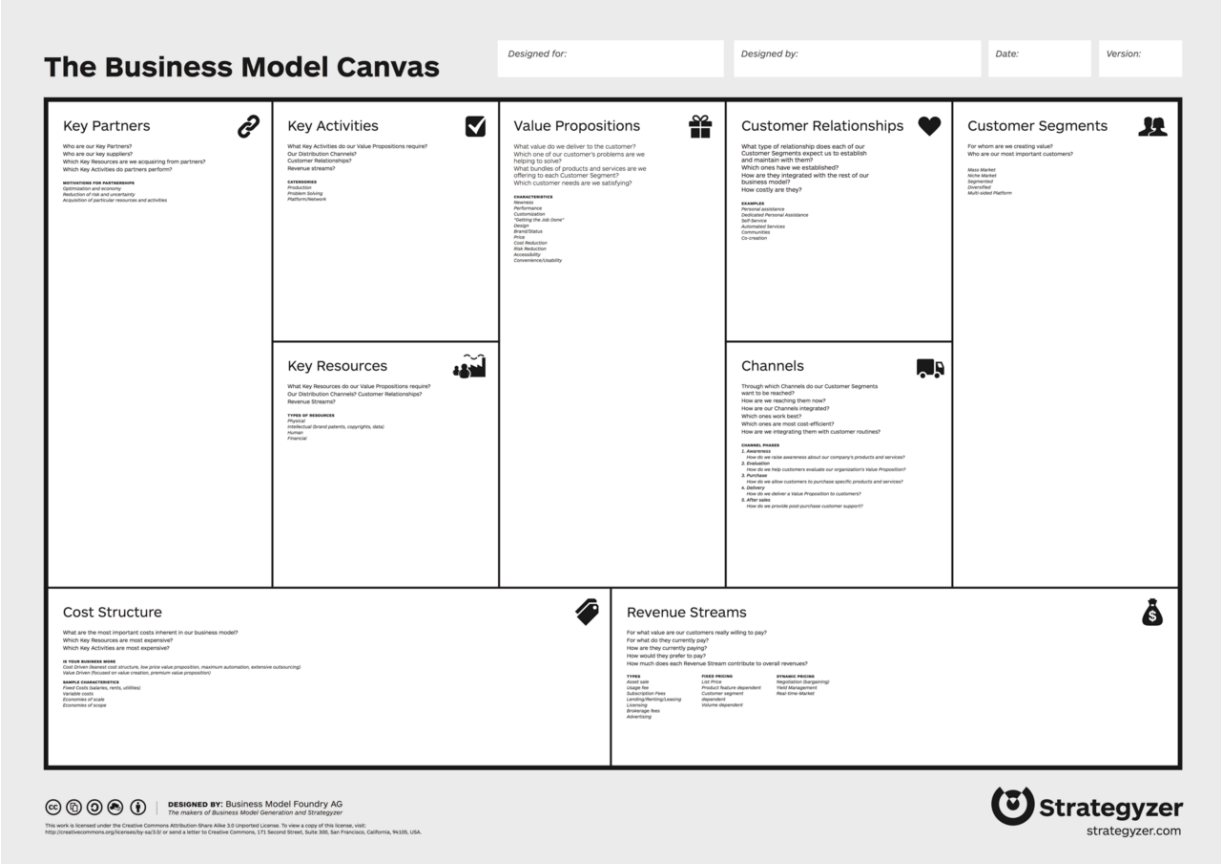


Figura 5.3: Business Model Canvas. Fonte: Wikipedia (https://pt.wikipedia.org/wiki/Quadro_de_modelo_de_neg%C3%B3cios).

O Canvas é apenas uma forma visual de organizar as informações, fácil e rápido de consumir. O planejamento de marketing costuma ter excesso de páginas — e, para falar a verdade, ninguém tem vontade de ler.

O que vimos é suficiente para perceber que não precisamos quebrar a cabeça para definir estratégia para plataformas. O que precisamos é dominar as especificidades da plataforma e nos ater aos conceitos centrais de estratégia:

- Quem usa/usará a plataforma?
- Quais recursos e funcionalidades podem ser disponibilizados?
- Como devem ser disponibilizados?
- O que pode ser acessado e por quem?
- Quais tipos de aplicativos/produtos podem ser construídos usando a plataforma?
- Como o público saberá sobre a plataforma?

Como não vamos estudar estratégia para um novo modelo de negócio, não seguiremos o padrão dos cursos de gestão de produto que usam o Business Model Canvas. Para responder a essas e outras perguntas, vamos usar o **Lean Canvas** como guia, já que o próprio criador do Lean Canvas, Ash Maurya (2022) diz: "*o Lean Canvas pode ser usado para descrever modelo de negócios, lançamento de produto ou até mesmo uma única funcionalidade*".

5.3 Lean Canvas para plataforma

Apesar de o Lean Canvas servir a diversos contextos, ainda não parece ideal para plataformas de uso interno. Você concorda que os frameworks existem para nos servir? Não somos nós que temos que servir a eles. Então, tomei a liberdade de adaptar o Canvas para realidade de plataforma.

O original possui o quadrante *Vantagem Injusta* (*Unfair Advantage*), "*... algo que não pode ser facilmente copiado ou comprado*" (MAURYA, 2022). A não ser que uma plataforma esteja longe de atender às expectativas, o mero fato de ela ser desenvolvida "dentro de casa" é uma vantagem injusta, se comparada a plataformas disponíveis no mercado.

Por isso, substituí *Vantagem Injusta* por ***Casos de Uso*** (*Use Cases*), para trazer o que foi mencionado anteriormente: *quais soluções a plataforma tornará possível que os times de produto desenvolvam*. A lista de casos de uso fundamentais revela o valor, já que plataformas de mercado, geralmente, acabam sendo genéricas demais, incapazes de atender a especificidades do negócio.

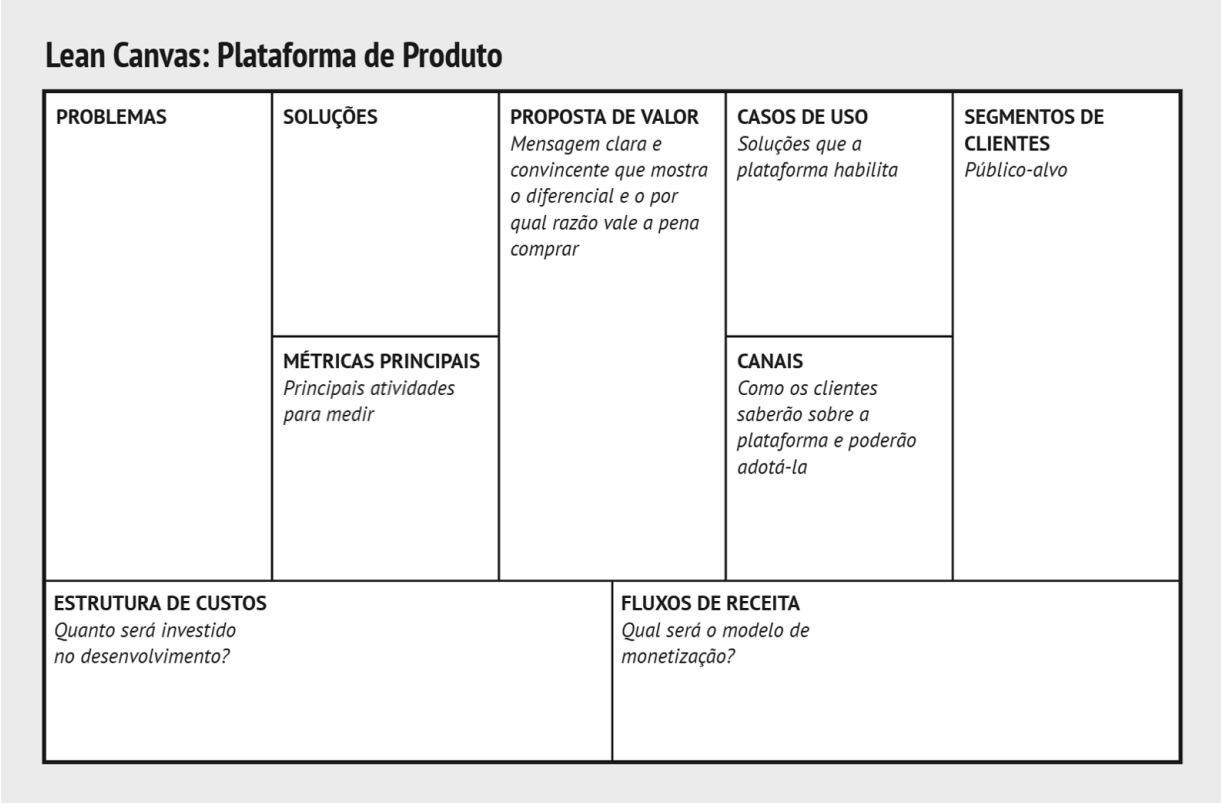


Figura 5.4: Lean Canvas para Plataforma.

Se no seu caso há alguma vantagem injusta da qual você pode se aproveitar, que precise validar ou manter no radar, talvez faça sentido manter o quadrante como Vantagem Injusta.

Também não se preocupe em preencher todos os quadrantes no dia zero. Aliás, se algum quadrante não fizer sentido, ignore, deixe em branco ou troque por outro importante ao seu contexto. Lembre-se: o objetivo não é preencher os quadrantes. **O objetivo é definir a estratégia**. O Canvas é apenas uma forma visual de organizar as informações.

Nos próximos capítulos, passaremos por cada quadrante do "novo" Lean Canvas. Para Ash Maurya, não existe uma ordem para preenchê-los, porém é pouquíssimo provável (para não dizer impossível) que se alcance o *product-market fit* sem antes encontrar o *problem-solution fit*. Para isso, é preciso avaliar atentamente os problemas que se apresentam para chegar às possíveis soluções — é por onde começaremos.

CAPÍTULO 6
Problemas e soluções

Apesar de tudo o que se ouve sobre *discovery* (identificar o problema antes de pensar em solução), é comum presenciar pelo mundo afora inúmeros casos de solução — supostamente genial, imaginada por algum stakeholder — desconexa de problemas, o que leva a exatamente ao oposto: *caçar problema para uma solução que já foi determinada*.

Isso também acontece para plataformas e vai continuar acontecendo. O quadrante *Problemas* em branco é um sinal. Por isso, o óbvio precisa ser dito: a primeira coisa a ser feita é preencher este quadrante respondendo à pergunta "*qual é o problema?*". Uma vez que o problema está visível, no quadrante *Soluções*, responde-se à pergunta "*como resolver o problema?*". Esses quadrantes trazem visibilidade à hipótese de *problem-solution fit*, cuja validação vem do processo de *discovery*.

Não se pode confundir com *product-market fit*. Dizer que o *problem-solution fit* foi alcançado significa que há evidências de que o problema é real e de que a solução encontrada realmente resolve o problema para determinado público-alvo — representada pela adoção dos *early adopters* (entusiastas e visionários). *Product-market fit* é a evidência de que existe demanda massiva, quando os *early majority* (pragmáticos) adotam a solução.

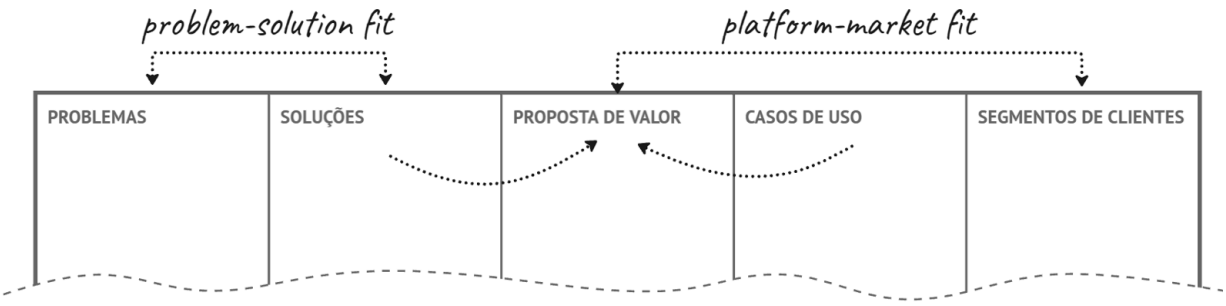


Figura 6.1: Problem-solution fit e Platform-market fit.

Então, qual é o *problema* que justifica a construção de plataforma?

Nas décadas mais recentes, a tecnologia avançou exponencialmente. O advento da internet, somado à popularização do celular, proporcionou mudanças de comportamento e oportunidades que viraram o mundo de cabeça para baixo. Isento dos limites físicos, o imaginário se torna real no digital. A qualquer momento, três adolescentes podem criar um software que, em poucos anos, pode se tornar um negócio "bilionariamente" lucrativo, desbancando empresas centenárias. Há 30 anos, esse fenômeno era inconcebível.

Se antes o capitalismo estimulava a competitividade agressiva, o que seria mais contundente do que competir agressivamente? Talvez inovar (ou copiar, por que não?). E rápido, ainda essa semana, no máximo esse mês. Mas como? Há tantos sofrimentos para tratar.

Usuários estão sofrendo. Reclamam que o produto não tem todas as funcionalidades de que precisam, que algumas delas não funcionam bem. Dizem que o sistema está lento, às vezes nem conseguem acessá-lo e demora horas para voltar. Compartilham toda essa frustração nas redes sociais. Ameaçam desinstalar o app e usar outro, já que é impossível obter o *job done* no momento em que precisam.

Devido a isso, a empresa está sofrendo. A saúde financeira apresenta riscos. Há queda na taxa de conversão de vendas. O custo com infraestrutura aumenta desproporcionalmente ao que se utiliza de fato.

Os times também estão sofrendo. São pressionados e não conseguem aproveitar oportunidades de mercado rapidamente, colocar experimentos no ar é demorado. Devs passam mais tempo corrigindo erros, aguardando testes e deploy do que desenvolvendo novas coisas. O ritmo de trabalho parece lento e improdutivo.

Sabe aquela história de "o que nos trouxe até aqui não é o que vai nos levar adiante"? É o que está acontecendo nesse cenário. A tecnologia que, até então, tinha custos baixos, desenvolvimento simples e rápido, é justamente o que está impedindo o crescimento e a escala do negócio. Essa *stack* tecnológica — combinação de framework front-end, framework back-end, bibliotecas e linguagens — se tornou incapaz de atender às demandas de um ambiente competitivo de ritmo acelerado.

Qual seria, então, o verdadeiro problema por trás desses que se apresentam?

6.1 Monolito é o problema

O cenário descrito nos conduz à causa raiz: *aplicação baseada em arquitetura monolítica*. O **monolito**, como é comumente chamado, é composto por três componentes principais que, juntos, formam **uma (apenas uma!) base de código**:

1. **Interface do usuário** (*user interface*): envolve o que é processado no lado do cliente (*client-side*), no navegador ou app do usuário final, além dos componentes de interface (UI) e as linguagens front-end.
2. **Lógica de negócio** (*business logic*): refere-se ao que se processa no lado do servidor (*server-side*) depois que recebe os dados do navegador ou app, aplicando as regras de negócio por meio de linguagens back-end.

3. **Interface de dados** (*data interface*): trata-se do código que interage com o banco de dados, como acesso, persistência e recuperação dos dados que preenchem a tela do usuário com as informações que estão no banco.

A base de código é conectada a um único banco de dados (*database*), o que algumas pessoas consideram como quarto elemento.

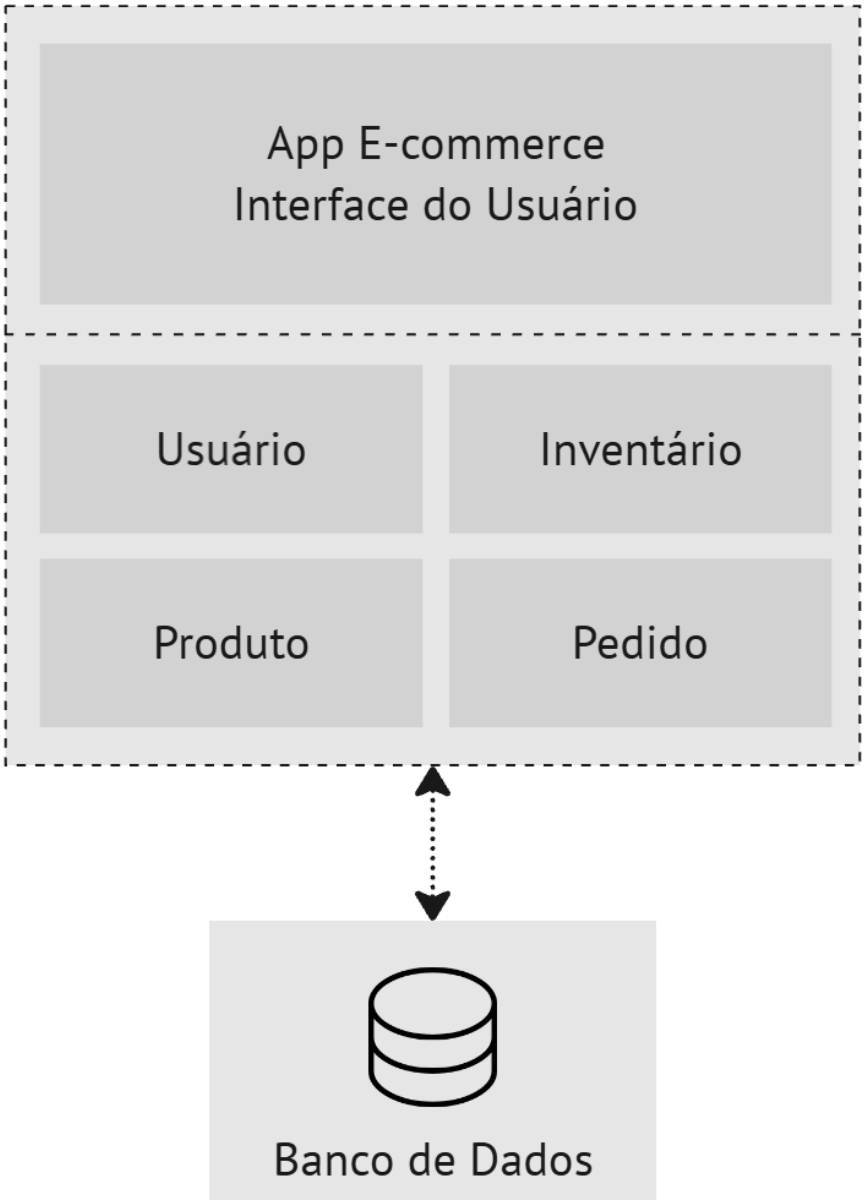


Figura 6.2: Monolito. Baseada em: IBM Developer (DESHPANDE, 2021).

Este formato simplifica e agiliza o desenvolvimento e, como único executável, é rápido de testar de ponta a ponta, fácil de *debugar* e de fazer deploy. É considerado o modelo tradicional de criação de produtos de software, além de ser a arquitetura recomendada no estágio de startup, por proporcionar a agilidade necessária para lançar o MVP, a fim de validar o produto e o modelo de negócio.

Monolito é eficaz, mas chega o momento em que o produto passa a ter tantas funcionalidades que aquela base de código que trouxe tantos benefícios vira o malefício. De tão gigante, fica complexo, inflexível (até rígido) e ainda tem de suportar alto tráfego e grandes volumes de dados. É quando os problemas emergem, e um puxa o outro, que puxa outro, e outro...

Perda de produtividade

A empresa contrata mais devs com a intenção de acelerar o desenvolvimento, e ocorre justamente o contrário. À medida que novas features são adicionadas, fica cada vez mais complicado ter muitos devs trabalhando na mesma base de código. Pode demorar meses para um dev que chegou recentemente na empresa se sentir confortável para contribuir em algo fora do trivial.

Muitas pessoas trabalhando no mesmo código tornam conflitos de código mais frequentes. Isso aumenta a propensão a falhas e o tempo para identificar a fonte dos problemas. No geral, os times gastam mais tempo resolvendo problemas técnicos do que desenvolvendo coisas novas e úteis aos usuários e ao negócio. Vivem mudando de contexto, "apagando incêndios".

A cada mudança, pequena ou grande, todo o monolito e suas dependências devem passar por testes e deploy em sua totalidade, o que pode levar horas.

Lentidão e indisponibilidade

Na arquitetura monolítica, todas as funções e processos são interligados e trabalham juntos. Uma interação do usuário faz toda a aplicação "pensar". Eventos que trazem tráfego intenso, como a Black Friday, deixam o app lento ou em modo não responsivo (*non-responsive*), tornando-se inutilizável.

Para ter uma noção do prejuízo da Amazon em 2018, olha só este trecho da notícia que a Bloomberg publicou: "*A Amazon.com Inc. iniciou a grande promoção Prime Day com falhas técnicas no site e aplicativo* (...). *Esperava-se que os compradores gastassem US$ 3,4 bilhões no site durante o evento, mais de 40% em relação ao ano anterior* (...)" (BLOOMBERG, 2018; tradução livre).

Alto custo e dificuldade de escalar

Para lidar com o cenário anterior, é necessário escalar a infraestrutura, mas não é possível escalar componentes individualmente, portanto sai mais caro. Ao invés de escalar apenas a parte que exige mais recursos, no monolito, "*se um processo do aplicativo apresentar um pico de demanda, toda a arquitetura deverá ser escalada*" (AWS, c2023).

Carregar todas as bibliotecas demora demais. Geralmente, leva dezenas de minutos para escalar o servidor e, somente depois disso, executa o sistema. Além disso, há limites para o quanto a aplicação pode escalar.

Limitação a uma stack tecnológica

O monolito é limitado às tecnologias já utilizadas. Se outra linguagem resolve melhor determinado problema, não pode ser adotada.

Para piorar, é comum monolitos dependerem de software desatualizado. O GitHub demorou quatro anos para atualizar para o Rails 3. Esse tipo de "atraso" pode limitar soluções e a falta de

atualizações de segurança pode trazer riscos severos ao negócio, então é preciso corrigi-las por conta própria.

Incapacidade de inovar

É difícil aproveitar e adaptar o que já existe a novos negócios. O código existente afeta e restringe qualquer novo código adicionado. Como todas as partes da aplicação são fortemente acopladas, uma alteração aparentemente inofensiva em uma parte pode ter consequências desastrosas em outra sem qualquer relação lógica, podendo derrubar todo o sistema. Esses riscos limitam experimentações e implementação de novas ideias. Por conta disso, é comum desenvolver a mesma funcionalidade, com certas diferenças, para novos produtos.

Em linha gerais, todo o processo fica lento. Afeta os times, o negócio e os clientes. A empresa precisa de uma solução que permita implementações rápidas, atualizações frequentes e maior escalabilidade. Qual é a solução que tem *fit* quando o monolito se torna problema?

6.2 Microsserviço é a solução

A solução que tem *fit* é a **arquitetura de microsserviços**. Ela foi idealizada, especificamente, para endereçar os problemas que surgem quando o monolito deixa de ser uma boa solução.

A lógica é a seguinte: se a fonte dos problemas é o fato de *todos os componentes (cliente e servidor) serem unificados em uma única base de código*, a solução mais óbvia é ir no sentido contrário, dividir o código gigante em códigos menores, formando serviços independentes que interagem entre si por meio de uma interface

(API), de modo a eliminar o acoplamento rígido entre as partes do monolito.

Assim, cada **microsserviço** é delimitado a uma única capacidade de negócio, cuja responsabilidade é executar as tarefas específicas desse **domínio**. Como exemplifica a imagem a seguir, o microsserviço *User* lida com as informações dos usuários, enquanto o microsserviço *Order* lida apenas com as do pedido. Como cada serviço tem finalidades diferentes, são colocados em servidores separados e cada um com seu próprio banco de dados.

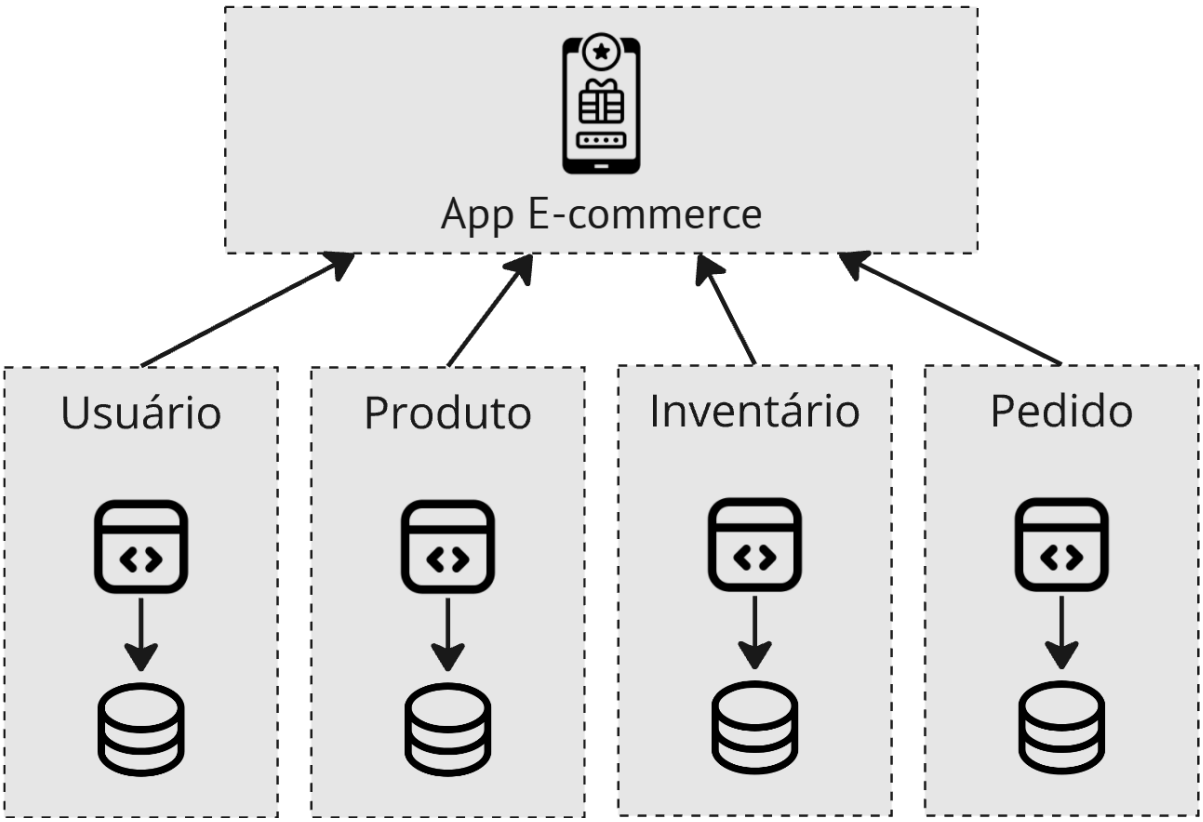


Figura 6.3: Microsserviços. Baseada em: IBM Developer (DESHPANDE; SINGH, 2021).

O princípio de **baixo acoplamento** minimiza as dependências entre os serviços e seus consumidores, o que permite alterações no back-end ou na composição do serviço, sem qualquer impacto aos consumidores da API. Neste paradigma, a integração direta com o

back-end é totalmente desaconselhada, pois impede essa autonomia.

Graças à independência entre os serviços, cada um pode ser desenvolvido na linguagem de programação mais eficaz para a solução. Com o seu próprio ciclo de vida de desenvolvimento, ele é atualizado, "*deployado*" e escalado sem afetar os demais serviços.

A abordagem de microsserviço também foi elaborada para lidar melhor com falhas. Além de ser mais rápido localizar o problema, isolá-lo e recuperar-se, quando um serviço cai, os outros continuam funcionais (exceto aqueles que são diretamente dependentes). É verdade que a experiência do usuário final fica limitada, mas parte do app funciona normalmente, o que é melhor do que todo o app ficar inacessível.

Por exemplo, o serviço de recomendação da Amazon, responsável por recomendar produtos que o cliente possa ter interesse, está fora do ar. No monolito, isso deixaria toda a Amazon inacessível. Nos microsserviços, o usuário acessa a página sem as recomendações. Apesar de não ser a experiência que a Amazon quer proporcionar, o site funciona e não deixa os usuários na mão.

O mesmo microsserviço pode ser reutilizado em mais de um processo, canal ou ponto de contato. Por conta disso, a quantidade de processamentos aumenta, então, para garantir performance e disponibilidade, é possível escalar somente os serviços que necessitam de mais recursos — como mostra a próxima imagem. Isso otimiza o custo de infraestrutura.

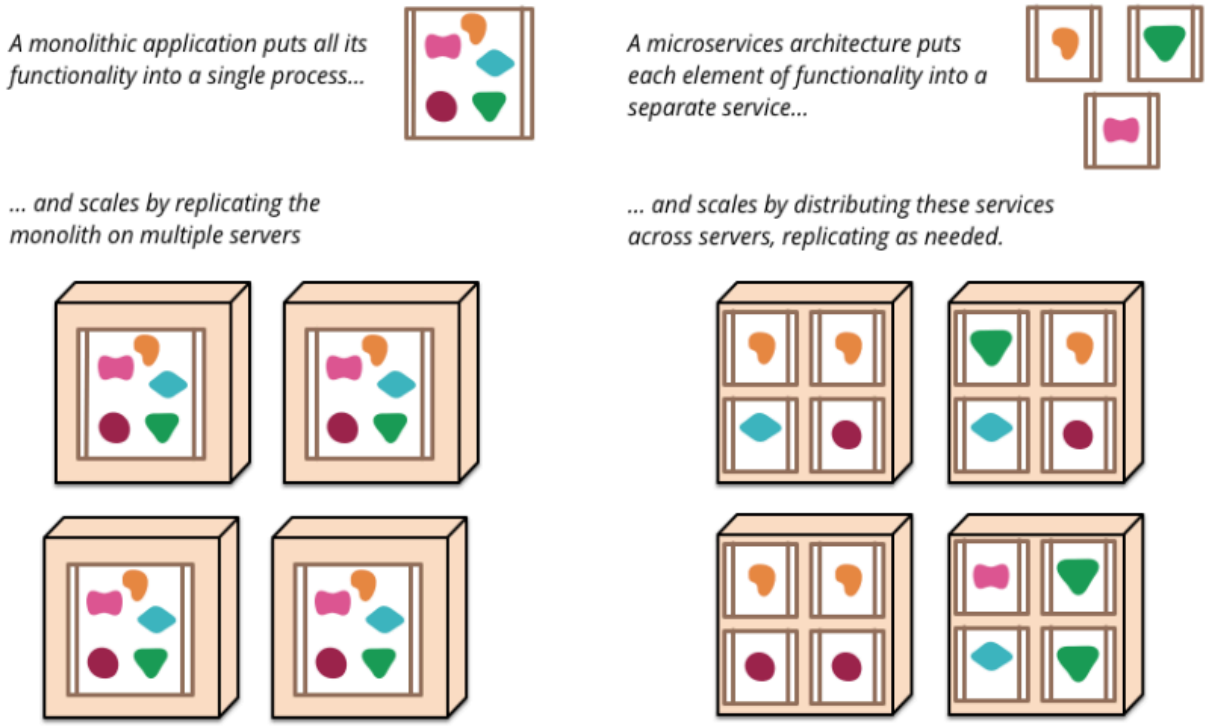


Figura 6.4: Monolito versus Microsserviços. Fonte: martinfowler.com (LEWIS; FOWLER, 2014).

Se o microsserviço crescer demais a ponto de aumentar a complexidade ou dificultar o gerenciamento, ele pode ser dividido em serviços menores.

Tudo indica que mudar de monolito para microsserviços é a solução para todos os problemas. Simplesmente perfeito! Será?

A arquitetura de microsserviços não é perfeita

Nada é perfeito, não é mesmo? Se você já trabalhou em ambos os tipos de arquitetura, já sentiu na pele a dor que é lançar coisas novas ou implementar pequenas melhorias no monolito. Já com microsserviços, sentiu o quanto de esforço é exigido para orquestrar as demandas junto aos times que dependem da plataforma, ou o inverso, conseguir que outros times absorvam demandas das quais a plataforma depende.

Difíceis de mensurar, os benefícios proporcionados pelos microsserviços vêm acompanhados de complexidade — não no microsserviço em si, mas na relação entre eles.

Quebrar a grande base de código de um monolito em microsserviços com códigos menores elimina a complexidade em relação ao código. A complexidade é transferida para o gerenciamento dos inúmeros serviços distribuídos: mais testes, deploys, *troubleshooting*, monitoramento, times.

É complicado conhecer todos os microsserviços disponíveis na empresa, entender como eles se relacionam, quem é responsável por determinado serviço e como evitar interferência em serviços dependentes. Isso exige que os times se comuniquem (muito) mais e colaborem entre si para coordenar atualizações e interfaces.

Escalar monolitos sai mais caro; por outro lado, é mais simples. Basta criar novas instâncias em vários servidores. Já escalar microsserviços exige monitoramento e gerenciamento cuidadoso dos recursos para garantir que sejam particionados e escalados adequadamente.

Monolito é suscetível a falhas generalizadas. Os microsserviços evitam esse tipo de ocorrência, porém, conforme aumenta a quantidade de pontos de comunicação, aumenta a quantidade de potenciais pontos de falha.

A implementação de mudanças que afetam vários serviços exige planejamento e coordenação de cada um dos serviços envolvidos — coisa que, no monolito, o time apenas altera os módulos correspondentes.

Por sorte (ou necessidade) inventaram a tecnologia em nuvem que lida com a maior parte da complexidade operacional, tornando-a visível e gerenciável, além de facilitar reutilização e escala.

6.3 Juntando tudo

Foi apresentado apenas o monolito como fonte dos problemas para não confundir com outros que emergem com o crescimento da empresa, já que a operação se torna naturalmente mais complexa e exige mais esforço de gerenciamento.

Se você é PM de uma plataforma com arquitetura de microsserviços, os problemas que você vivencia talvez sejam diferentes dos apontados. É mais comum passar por problemas como os que vimos na seção *A arquitetura de microsserviços não é perfeita*: latência na comunicação entre os serviços e incidentes devido a um ou outro ponto de falha.

De qualquer forma, até que outras soluções sejam inventadas, a arquitetura de microsserviços é o estado da arte para tratar sistemas complexos e em contínua evolução. Há efeitos colaterais, como outras complexidades e desafios de implementação. Microsserviços implementados inapropriadamente podem trazer os mesmos problemas que o monolito, potencializados pela complexidade de gerenciar serviços distribuídos.

Por isso, o design da arquitetura e a escolha das tecnologias são cruciais para o sucesso da plataforma. O resultado disso deve ser um sistema altamente resiliente, robusto, escalável, flexível e extensível, para que seja fácil de manter e seja possível adicionar e remover casos de uso — essas características são abordadas em detalhes no capítulo *Proposta de valor e casos de uso*.

Chegar a essa visão pode levar anos e gera ansiedade generalizada para obter retorno sobre o investimento. Nosso papel enquanto PMs é comunicar os problemas com clareza, o que os está causando e os benefícios que se esperam das soluções — o *problem-solution fit*.

Com isso em mente, preenchemos os quadrantes *Problemas* e *Soluções* do Canvas.

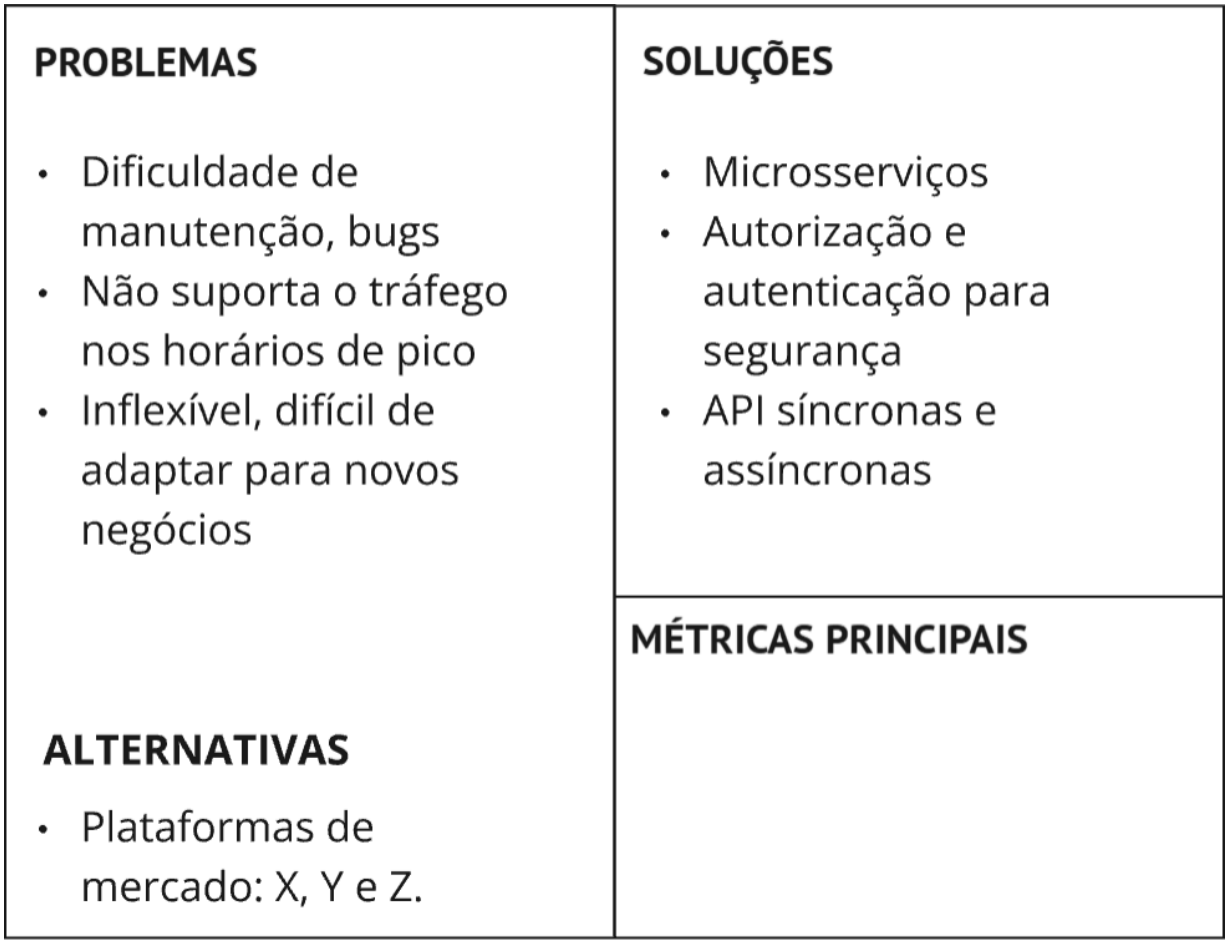


Figura 6.5: Lean Canvas — Problemas e Soluções.

Note que vimos meramente o aspecto tecnológico. Na hora de montar o seu, lembre-se de adicionar os problemas que as funcionalidades (ou a falta delas) estão causando.

Por exemplo, uma unidade de negócio da empresa quer enviar e-mails promocionais, mas a plataforma só dispara e-mails transacionais — o envio é disparado automaticamente quando o sistema recebe determinada informação. O quadrante *Problemas* conteria algo como: "A unidade de negócio X envia e-mails promocionais manualmente na data definida no cronograma das campanhas. Além de ineficiente, estima-se que há perda de 20% de conversão por erros manuais e atrasos no disparo". E uma hipótese a ser apontada em *Soluções* poderia ser: "Agendamento para disparo de e-mails". Adiantando o quadrante que veremos mais à

frente no livro, *Casos de Uso*, essa feature poderia ser usada para: "Envio de e-mails que seguem alguma forma de programação: promocionais, comunicados, newsletters, régua de comunicação".

No quadrante *Problemas* há a opção de listar *Alternativas* que os clientes costumam usar ou que fornecem o mesmo tipo de solução — os *concorrentes*. Conhecer as alternativas disponíveis no mercado é excelente para comparar e saber o que se espera que a sua plataforma faça. Quando comecei a gestão da Order Platform, uma das primeiras coisas que fiz foi procurar na internet plataformas que processam pedidos. As que mais absorvi conhecimento foram a Square (SQUARE DEVELOPER, c2021) e o Google Shopping (GOOGLE DEVELOPERS, c2021). Com esse *benchmark*, eu pude entender as funcionalidades que plataformas de pedidos costumam ter, de que forma o processamento de pedidos foi tecnicamente resolvido e, principalmente, como comunicar o valor sem usar termos técnicos.

Muito bem, entendemos os problemas e levantamos hipóteses de solução, mas você sabe que de nada adianta a solução ter *fit* perfeito para o problema se ninguém a usa. É preciso identificar *quem* enfrenta os problemas e como essa pessoa saberá que a solução existe. Esses são os assuntos do próximo capítulo.

CAPÍTULO 7
Segmento de clientes e canais

Qualquer solução resolve um problema para alguém. O quadrante *Segmento de Clientes* serve para trazer visibilidade do público que enfrenta o problema, em outras palavras, quem é o *target* da solução.

Essas pessoas precisam saber que existe uma solução para o problema. Para que isso aconteça, é necessário haver meios de alcançar o público-alvo e atraí-los para aquisição da solução. Esses meios são listados no quadrante *Canais*.

É provável que, em algum momento da jornada, o usuário final interaja com alguma funcionalidade fornecida pela plataforma. Portanto, para falar sobre segmentos de clientes, é indispensável considerar a **cadeia de valor** — processos que, juntos, viabilizam a entrega de valor ao cliente final.

7.1 Plataforma na cadeia de valor

Em empresas que fabricam produtos físicos, a cadeia começa na matéria-prima e termina com a venda do produto acabado aos consumidores. Para empresas que criam produtos digitais, a cadeia de valor começa em suas plataformas e vai desde o back-end até quem interage com o app (o cliente final), conforme mostra a figura adiante.

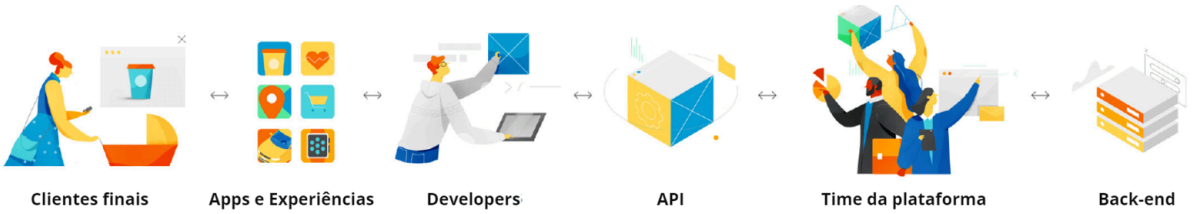


Figura 7.1: Cadeia de valor. Adaptado de: The API Product Mindset (GOOGLE CLOUD, 2018).

Começando da direita para a esquerda, vem o **back-end**, onde se encontram os *recursos* — qualquer coisa que a empresa possui e vê valor em compartilhar: dados, trechos de código, software ou serviços. Esses recursos residem no servidor e no banco de dados — estrutura já vista em capítulos anteriores.

Em seguida vem o **time da plataforma**, que cria o back-end e disponibiliza uma interface, a **API**. Assim, no estágio seguinte, os recursos ficam acessíveis para uso — internamente pelos times e/ou externamente por parceiros ou devs terceiros, chamadas respectivamente de API privada, API para parceiros e API externa (já abordado no capítulo *Anatomia da plataforma*). As APIs atuam como filtro e revelam apenas o que o time da plataforma deseja que seja acessado, já que nem todo conteúdo que está no servidor deve ser aberto ao exterior.

Depois vem a pessoa **developer**. Ela encontra a API e a usa para criar o **app**. Aqui reside a entrega de valor da API. Os componentes habilitam novas formas de usar os recursos e a reutilização. Em vez de repetir o trabalho que já foi feito, permite que os times trabalhem mais rápido e até melhor. Com o app pronto para fornecer novas **experiências**, ele é disponibilizado na app store, onde o **cliente final** pode fazer a instalação para utilizá-lo.

Veja que as APIs tornam os recursos compatíveis com variados dispositivos e sistemas operacionais. Assim, os usuários finais têm a liberdade de acessar os apps entre dispositivos em uma experiência integrada, *omnichannel*.

Resumindo em um exemplo real, devs usam a API de status dos pedidos da Order Platform para exibi-los nos apps do iFood. O usuário final, por sua vez, acessa o app (ou a versão web) em qualquer dispositivo quando deseja saber em qual status está o andamento do pedido. A informação é a mesma em todos os dispositivos.

A cadeia expõe que a plataforma impacta as experiências de dois perfis de pessoas: *developers* dos apps — produtos *customer-facing* e também podem ser outras plataformas — e usuários dos apps (clientes finais). Seriam ambos clientes da plataforma?

7.2 Clientes e usuários da plataforma

Antes de refletir sobre a pergunta, vamos relembrar um conceito básico: *cliente* é quem compra o produto; quem o usa é *usuário*. Quando os pais compram brinquedos para os filhos, os pais são os clientes, enquanto as crianças são as usuárias dos brinquedos. É comum o cliente ser o usuário também, como quando uma pessoa compra um xampu e é ela quem o usa.

Plataforma de produto pode ter como alvo clientes internos à empresa, externos ou ambos. A abordagem deste livro é a plataforma como recurso interno, ou seja, é desenvolvida dentro da empresa e **o cliente é a própria empresa**, geralmente representada na figura do CTO, times dos apps e, às vezes, por outros stakeholders internos. Pouca coisa muda quando a plataforma é voltada a clientes externos. A diferença é que existe um processo de decisão de compra que costuma envolver mais stakeholders, como jurídico, financeiro e compras.

Voltando à cadeia de valor, vimos que devs usam as APIs para adicionar funcionalidades da plataforma ao app que criam. Isso mostra que **as pessoas developers são usuárias da plataforma**

e, por fazerem parte de times interessados na solução, **os times dos apps são clientes**: product managers, product designers e pessoas de engenharia — engineering managers, tech leads e, inclusive, developers. Estes formam o público imediato.

> Do todo da plataforma, a API é a parte usável e comercializável. Por isso, *plataforma* é comumente chamada de **produto de API** (*API product*). Portanto, além de cargos como technical product manager e platform product manager, você pode se deparar também com API product manager.

Já os usuários finais, que usam o app (que é alimentado pela API), são atendidos indiretamente e, apesar de secundário, é o público que motiva muitas das decisões, desde funcionalidades até requisitos legais, como adequação à LGPD e permissões de quais aplicações poderão ou não acessar dados sensíveis.

Disponibilidade, performance e segurança da plataforma têm impacto direto na experiência do usuário final. Se os clientes dos nossos clientes tiverem má experiência, certamente os nossos ficarão igualmente insatisfeitos. Por isso, é importante entender o suficiente sobre as necessidades dos usuários finais junto ao time do app, assim o time da plataforma pode obter *insights* que ajudem a melhorar a experiência do usuário final e a elaborar APIs que capacitem os devs do app a criar essas experiências.

Naturalmente, os devs usuários têm problemas para resolver, e a plataforma existe para apoiar no *job-to-be-done*. Quase sempre os times sofrem por falta de recursos e/ou de tempo, então optam por buscar algo pronto (dentro ou fora da empresa) que ajude as pessoas devs desenvolverem mais rápido e mais fácil. A escolha dependerá *do que* a plataforma fornece e de *como* fornece.

Para que a escolha seja a sua plataforma em vez que alguma disponível no mercado, como PM, o seu trabalho é oferecer,

primeiro, uma proposta de valor ao cliente; segundo, proporcionar uma boa experiência aos devs usuários.

A plataforma só traz valor e experiências se for usada. Para ser usada, ela precisa ser encontrada. Para ser encontrada, precisamos disponibilizar um **canal** e divulgá-lo.

7.3 Canais de entrega de valor

A jornada do cliente inicia pela busca por informações sobre a solução para que consigam avaliar se tem *fit* com o problema que precisam resolver. No mundo das APIs, geralmente, todo esse conteúdo é centralizado em um canal chamado **portal do desenvolvedor** — mencionado brevemente no capítulo *Gestão do ciclo de vida*.

Clientes em geral querem saber o que a plataforma fornece sem adentrar em detalhes técnicos de como usar a API. Já devs, como usuários, têm necessidades mais específicas. Eles têm interesse na documentação da API para entender como usá-la, mecanismos de segurança, tratamento de erros, *sandbox* para testar e integrar na API, SDKs, dados de teste etc.

Como PMs, nosso papel é garantir que, ao acessarem o portal, os clientes internos consigam identificar o que é a plataforma, suas funcionalidades, os casos de uso que ela viabiliza e como começar a usar. É o tipo de conteúdo *marketeiro*, que ajuda a "vender" a plataforma, como mostra o *print* a seguir, do portal do desenvolvedor do Twitter (https://developer.twitter.com/en). Repare na chamada principal: "*seconds to 'hello world*" (segundos para o "olá, mundo"). Levar segundos para obter algum resultado da API é um apelo que certamente deixa os clientes felizes.

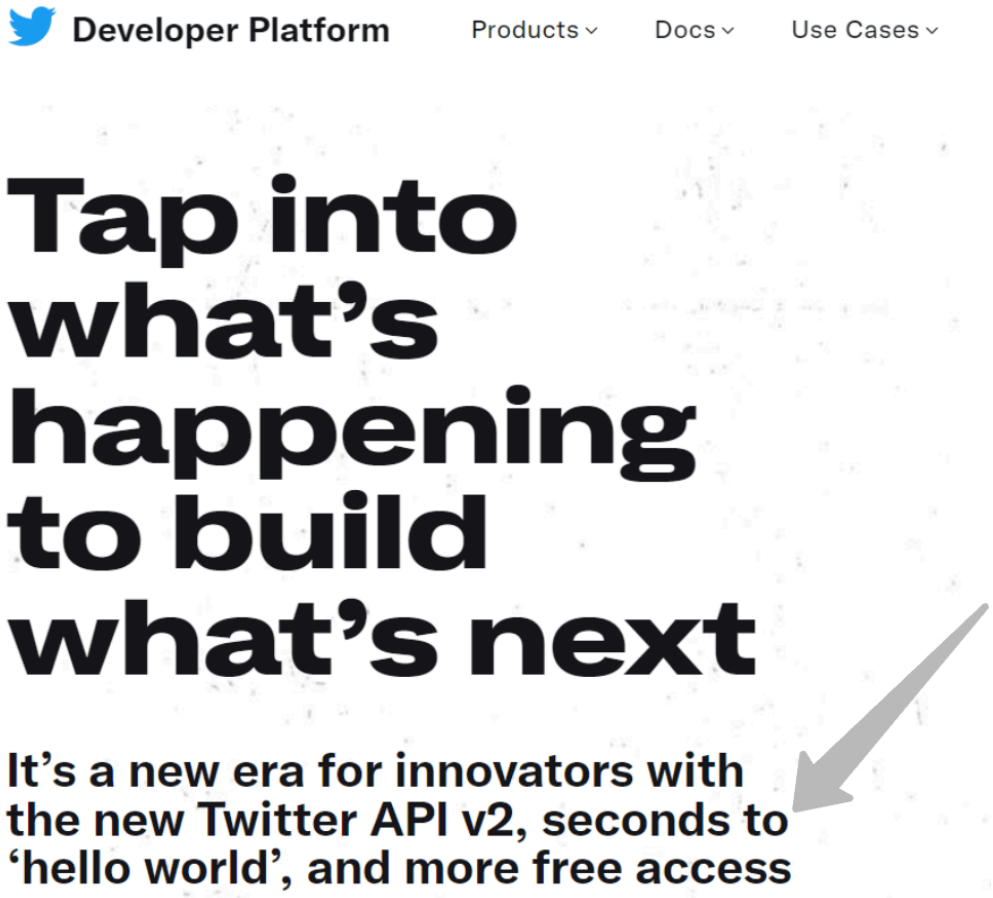


Figura 7.2: Fonte: Twitter Developer Platform.

> Nunca ouviu falar em "hello world"? Veja a definição no Wikipedia: *Programa Olá Mundo*. https://pt.wikipedia.org/wiki/Programa_Ol%C3%A1_Mundo

Pensando em facilitar a adoção, o ideal é que o processo seja simples e objetivo, incluindo preenchimento de formulários, configuração do ambiente e obtenção das credenciais. Não pode faltar, também, a documentação com o nome das APIs, métodos, argumentos e valores de retorno — vamos nos aprofundar nas Partes 3 e 4. Melhor ainda se tiver trechos de código que os devs possam usar para iniciar o projeto deles, como do *print* adiante, também do portal do desenvolvedor do Twitter.

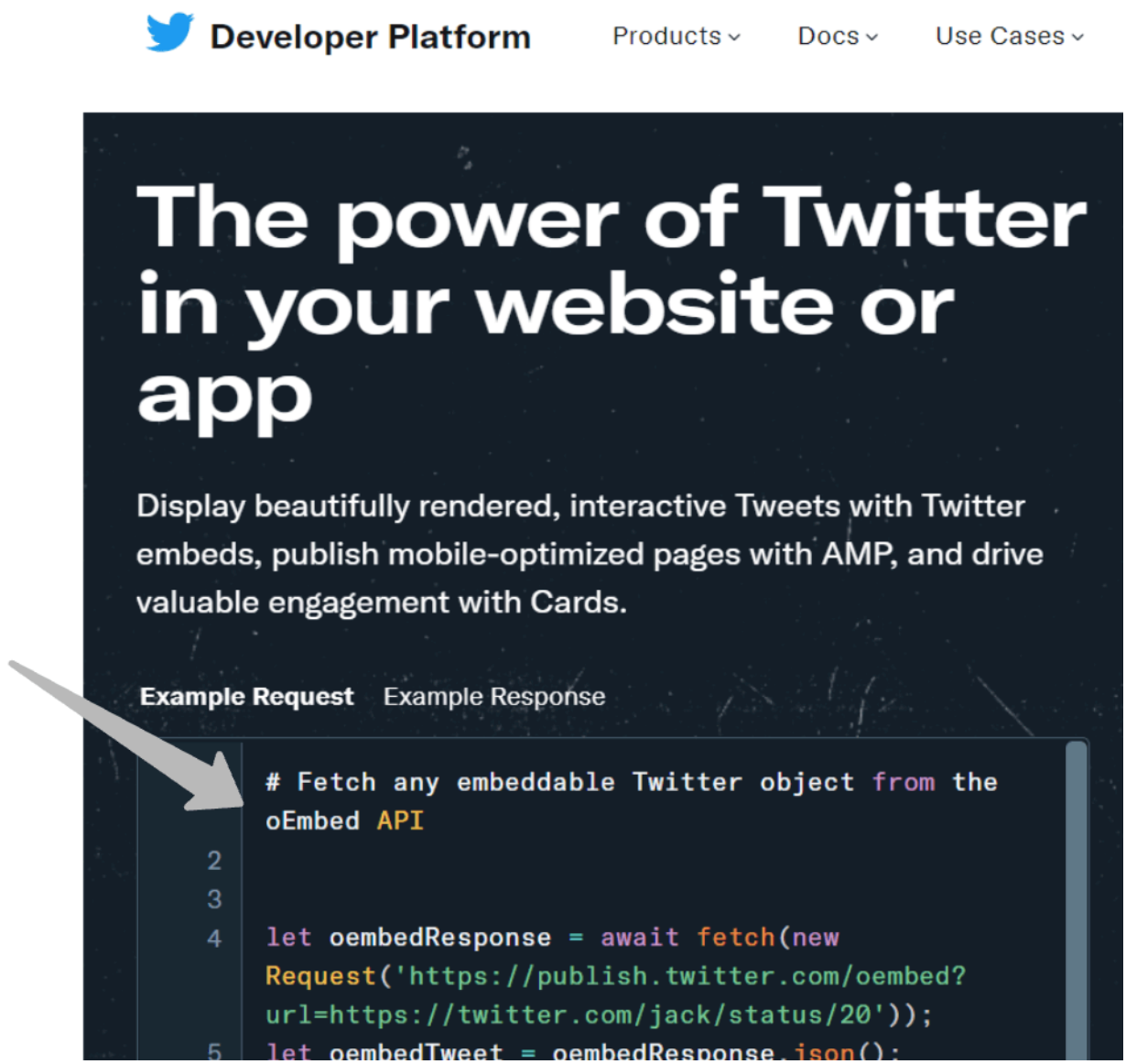


Figura 7.3: Fonte: Twitter Developer Platform (https://developer.twitter.com/en).

Ainda no portal do desenvolvedor, pode ser conveniente ter uma seção de “perguntas frequentes” (FAQ) ou uma seção "como fazer" com o passo a passo de como devs podem realizar tarefas comuns. No melhor dos mundos, o portal possui exemplos e trechos de código em diferentes linguagens de programação e ambiente de testes confiável.

O canal é o meio usado para persuadir a audiência sobre o valor do produto, onde se exibe que vai resolver determinados problemas e, claro, um caminho fácil e conveniente para o cliente adquirir o produto. Aquisição e ativação levam a um KPI típico de produto, número de clientes. E, para contribuir com o crescimento da base, entre outras coisas, PMs atuam na experiência do usuário, tirando fricções na fase de adoção. Por que seria diferente quando usuários são devs? A experiência do developer (*developer experience* — DevX ou DX) é mencionada com frequência na terceira parte do livro e é tratado em profundidade no capítulo *Da ideação ao teste de usabilidade*.

7.4 Juntando tudo

Em geral, produtos têm menos variação de *job-to-be-done* do que plataformas. Ambos têm que lidar com diferentes usuários que querem diferentes funcionalidades, mas a percepção de valor do produto é mais coesa se comparado à plataforma.

Plataformas atendem a contextos variados que compartilham a mesma atividade de base e é difícil conseguir alcançar o *fit* perfeito a todos os contextos. Por exemplo, os consumidores do iFood querem saber o status do pedido, contudo, o fluxo, os processos de produção e a entrega da refeição são diferentes de compra de mercado. Ambas as unidades de negócio precisam do mesmo tipo de informação, no entanto, algumas informações são distintas. Como lidar com esse tipo de incompatibilidade será um dos assuntos do próximo capítulo, mas adiantando um pouco, é preciso um certo nível de adequação à necessidade de cada app.

Quanto ao canal e à experiência das pessoas devs, eu diria que devemos considerar com parcimônia o que disponibilizar de documentação e ferramental. Os times vão integrar à plataforma apenas uma vez. Se a plataforma tem três potenciais clientes, não

há necessidade de um portal do desenvolvedor típico, nem exemplos de código em linguagens diferentes. Uma ferramenta simples, como o Atlassian Confluence, o Notion ou o Google Docs, é suficiente.

É uma boa prática exemplificar a implementação com trechos de código em linguagens diferentes, mas quando se usa várias linguagens dentro da organização e o número de potenciais clientes é pequeno, faz sentido exemplificar apenas com a mais usada entre esses clientes.

Se a plataforma tem dezenas de APIs e/ou dezenas de times a integrar, talvez uma ferramenta para documentar e testar, como o Swagger, Apiary ou Postman, já atenda às necessidades. Também tem o WordPress, que pode ser instalado em um servidor comum e é excelente para gestão de conteúdo. Além de possuir plugins do Swagger, é possível transformar o WordPress em um verdadeiro portal do desenvolvedor com o Tyk.

Há ainda os gerenciadores de API, que vimos no capítulo *Gestão do ciclo de vida*. Eles oferecem o portal do desenvolvedor com tudo o que tem direito. Se as APIs já estão hospedadas nesse tipo de sistema, não tem o que pensar: é só usar a ferramenta de portal.

Uma vez que você sabe quem são os times clientes e definiu os meios pelos quais a audiência vai acessar o conteúdo e as ferramentas da sua plataforma, é hora de preencher os quadrantes *Segmento de Clientes* e *Canais* do Canvas.

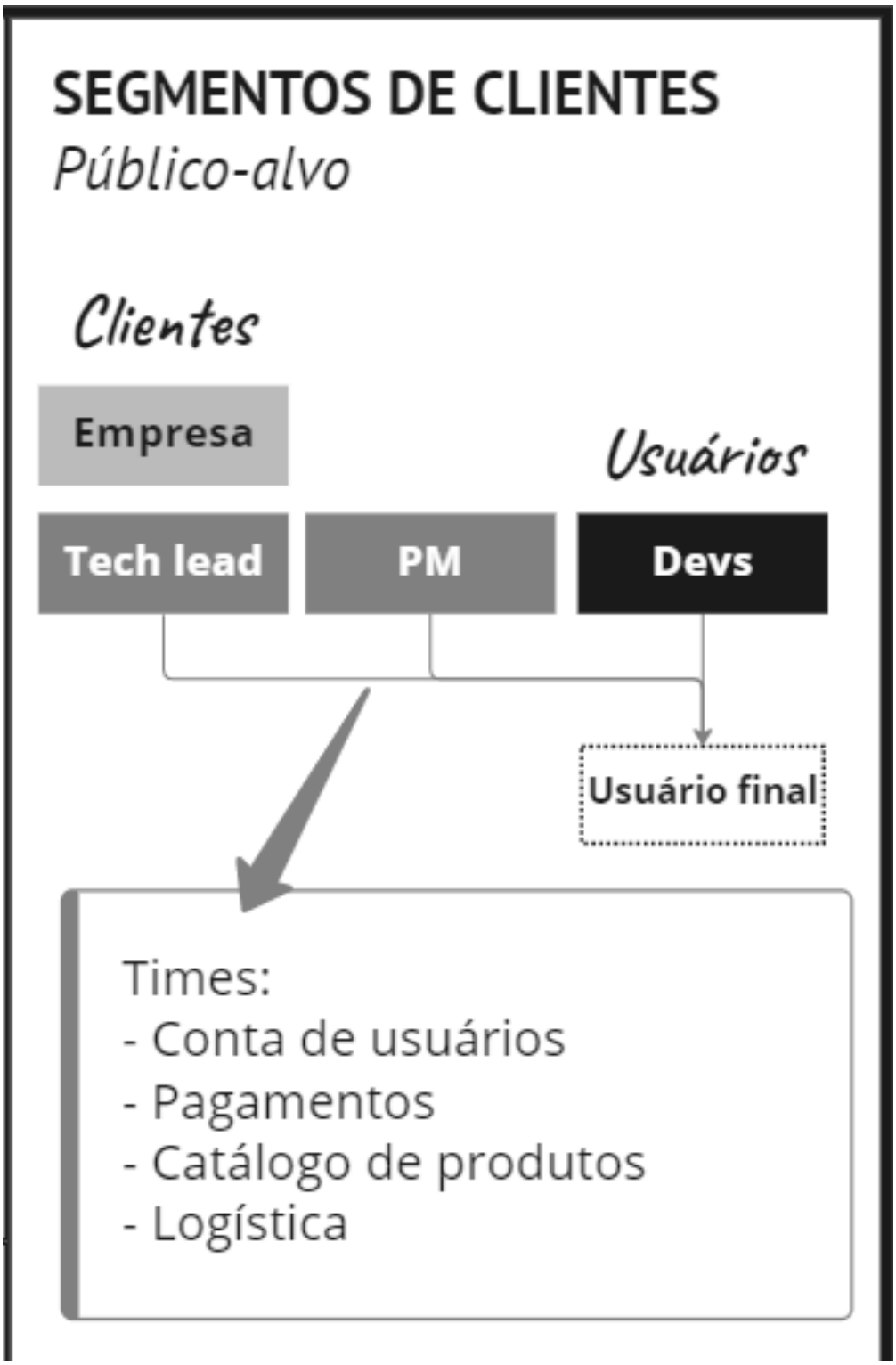


Figura 7.4: Lean Canvas — Segmentos de clientes.

É comum cada app proporcionar experiências diferentes a seus usuários finais. Por isso, vale a pena listar os times, porque cada um pode precisar de alguma forma de adaptação da plataforma.

É provável que cada perfil de stakeholder passe por problemas diferentes, acesse canais e obtenha propostas de valor distintos, por isso, é uma boa destrinchar cada perfil e separar por cor, como na próxima imagem. Dessa forma, fica visível a relação entre os itens dos demais quadrantes e cada perfil de cliente/usuário.

Figura 7.5: Lean Canvas — Canais.

Que tal aproveitar para atualizar os quadrantes *Problemas* e *Soluções* relacionando os itens aos respectivos clientes/usuários? No Canvas de exemplo, os itens sem marcação referem-se aos problemas e soluções relacionados à própria plataforma, por isso,

não estão associados a nenhum público do quadrante *Segmento de Clientes*.

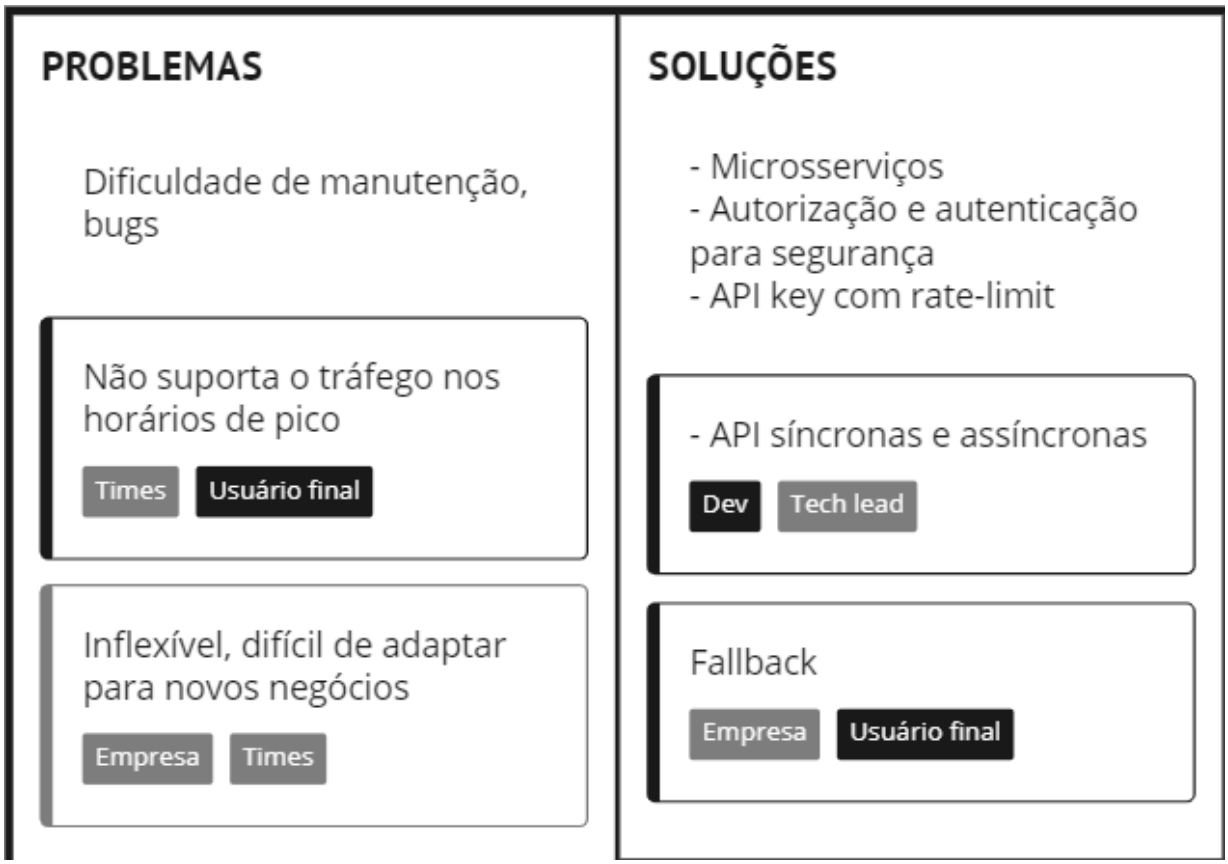


Figura 7.6: Lean Canvas — Problemas e Soluções.

Depois de entender o problema e definir o *fit* de solução que vai trazer benefícios ao público-alvo, chega-se à proposta de valor. Em capítulos anteriores, já vimos um pouco dos benefícios que trazem a própria natureza das tecnologias; no próximo capítulo, *Proposta de valor e casos de uso*, vamos explorar isso melhor.

Capítulo 8
Proposta de valor e casos de uso

A proposta de valor consiste em um conjunto de benefícios que prometemos entregar. Com o segmento de clientes em mente listamos quais benefícios são esses no quadrante *Proposta de Valor* do Canvas.

Em algumas circunstâncias, mostrar como a plataforma se relaciona com a estratégia do negócio é uma tarefa difícil e, se nem as pessoas do time da plataforma veem valor, imagine stakeholders de negócios. É compreensível que pessoas de negócios não entendam o valor de uma plataforma, mas pessoas de engenharia!? Por que será?

Em 2022, a Forrester Consulting publicou uma pesquisa feita com 311 tomadores de decisão de TI e negócios. A maioria das empresas participantes dessa pesquisa possui mais de 100 APIs, contudo, "*mais da metade dos entrevistados diz que as APIs nunca — ou raramente — são reutilizadas e que muitas APIs internas têm apenas um ou dois usuários*". Essa baixa adoção é o motivo pelo qual 71% deles "*concordam ou concordam fortemente que não obtiveram os resultados de negócio que planejaram para as APIs*". A pesquisa ainda chama a atenção para o fato de que "*dois terços dizem que o valor das APIs não é percebido até que sejam amplamente utilizadas*". (FORRESTER CONSULTING, 2022; tradução livre).

No capítulo *O que é plataforma, afinal?* vimos que o estudo do Dr. Bruce Cameron sobre plataforma de produtos explica que a estratégia de plataforma traz três benefícios centrais: redução do *time-to-market*, dos custos e dos riscos. O mesmo estudo revela que a maioria das empresas falha em obter retorno sobre o que foi investido nas plataformas e ele atribui isso a quatro fatores (EASTWOOD, 2016; tradução livre):

1. Forçar o desenvolvimento de plataforma para um único produto;
2. Negligenciar a articulação do valor do compartilhamento (a reutilização);
3. Avaliar retornos prematuramente;
4. Tentar mudar para plataformas sem realizar as mudanças organizacionais necessárias.

Lembra que a estratégia de plataforma é a reutilização de recursos existentes na empresa? Se a plataforma não é reutilizada, não há retorno sobre o investimento. Se não há retorno sobre o investimento, ninguém vê valor, mesmo.

Uma saída é evitar que os fatores citados pelo Dr. Cameron ocorram. Como PMs, não temos autoridade para atuar nos quatro itens — no máximo, podemos influenciar decisões —, mas é nossa responsabilidade evitar dois deles:

- "*Forçar o desenvolvimento de plataforma para um único produto*": garantir que a plataforma seja reutilizável e seja utilizada por mais de uma aplicação.
- "*Negligência ao articular o valor do compartilhamento*": traduzir recursos técnicos complexos em proposta de valor para o negócio, stakeholders e usuários.

Mais adiante trataremos esses dois itens. Primeiro, vamos refletir sobre a relação entre tecnologia e negócio.

8.1 A relação entre tecnologia e negócio

Em empresas nativas digitais, não existe separação entre tecnologia e negócio. Várias empresas nem possuem local físico para recepcionar clientes. O único meio de interação é o app. Logo, sob o ponto de vista do cliente, **a empresa é o app**.

Diante disso, a tecnologia é o *core business*. Sem ela, a empresa não existe. Esse conceito é tão forte que essas empresas são chamadas de "empresas de tecnologia" ou "*qualquercoisatech*": *fintech*, *edtech*, *foodtech*, *healthtech*, *logitech* — empresas que não nasceram na era da economia digital, mas, ao passarem pela famosa transformação digital, acabam se tornando empresas de tecnologia.

Se a tecnologia e o negócio são partes intrinsecamente ligadas e indissociáveis, significa que **o negócio é tão forte quanto a tecnologia que está por trás dele**. Por isso, arrisco contrariar a norma de que proposta de valor não é sobre tecnologia, defendendo que *também é sobre tecnologia*. Algumas pessoas vão discordar se avaliarem sob a perspectiva técnica, por exemplo, quando se pretende conectar dois sistemas para troca de informações entre si. Neste caso, a API é essencialmente uma questão de **arquitetura de software** (*software architecture*): *como* integrar os sistemas da melhor forma.

Sob a perspectiva de valor, APIs vão além de integrações, são a **transformação do valor do negócio em ativos técnicos**, componentes reutilizáveis capazes "(...) *de conectar unidades de negócios para expor funcionalidades e conhecimentos importantes de forma que ajude a empresa a ser efetiva. As APIs costumam ser uma maneira de desbloquear o valor que já existe na organização, por exemplo, por meio da criação de novos aplicativos, permitindo novos fluxos de receita e iniciando novos negócios*" (MEDJAOUI et al., 2018; tradução livre). Nesse sentido, plataformas são artefatos de **arquitetura de negócios**: ajudam a empresa a alcançar os objetivos, a ser competitiva no mercado e a se preparar para enfrentar desafios e oportunidades.

Não pense que arquitetura de negócios é mais importante do que arquitetura de software. Ambas são diretamente ligadas. A arquitetura do software leva à arquitetura do negócio. O design da arquitetura é o que torna a plataforma reutilizável; ser, de fato, usada depende do quão útil ela é para os clientes internos.

8.2 O valor depende do design da arquitetura

Reutilização é o motivo de a plataforma existir. Posicionar os componentes como ativos valiosos para o negócio implicitamente os promove como adaptáveis para uso em múltiplos contextos, em mais de um processo, em diferentes canais de negócios ou pontos de contato.

Isso significa que plataformas são desenvolvidas para atenderem ao cenário atual do negócio, vislumbrando sua utilidade no longo prazo: absorver cenários previstos e estar preparada para cenários desconhecidos, até mesmo para inesperadas fusões e aquisições de negócios externos (*M&A — Mergers and Acquisitions*). Para que a plataforma se mantenha coerente à promessa da proposta de valor a longo prazo, ela precisa ser projetada para escalar de acordo com o crescimento da empresa e ser adaptável a novos contextos.

O cumprimento ou não da promessa feita na proposta de valor depende de se fazer as escolhas certas em relação à arquitetura no início do desenvolvimento da plataforma: tecnologias, padrões, ferramentas e processos que serão usados para desenvolver e manter a plataforma. A arquitetura é a base para garantir que os componentes (*building blocks*) e suas APIs não sejam meras interfaces entre sistemas, mas, sim, interfaces a uma proposta de valor.

Novas soluções e modelos de negócios

A combinação de APIs de diferentes plataformas capacita a criação de novas soluções ou, até mesmo, modelos de negócio inusitados — eventualmente, a menor custo, mas certamente, com menor esforço.

Por exemplo, vejamos a tela inicial do Google Maps. Como não trabalhei no Google, imagino que a barra de pesquisa use a tecnologia de pesquisa do Google. O *widget* que exibe o clima deve ser alimentado por alguma empresa que fornece dados

meteorológicos (API externa). O anúncio utiliza os recursos do Google Ads. Em resumo, por meio de APIs de outros serviços, o time do Google cria uma solução que proporciona uma experiência diferenciada ao usuário final.

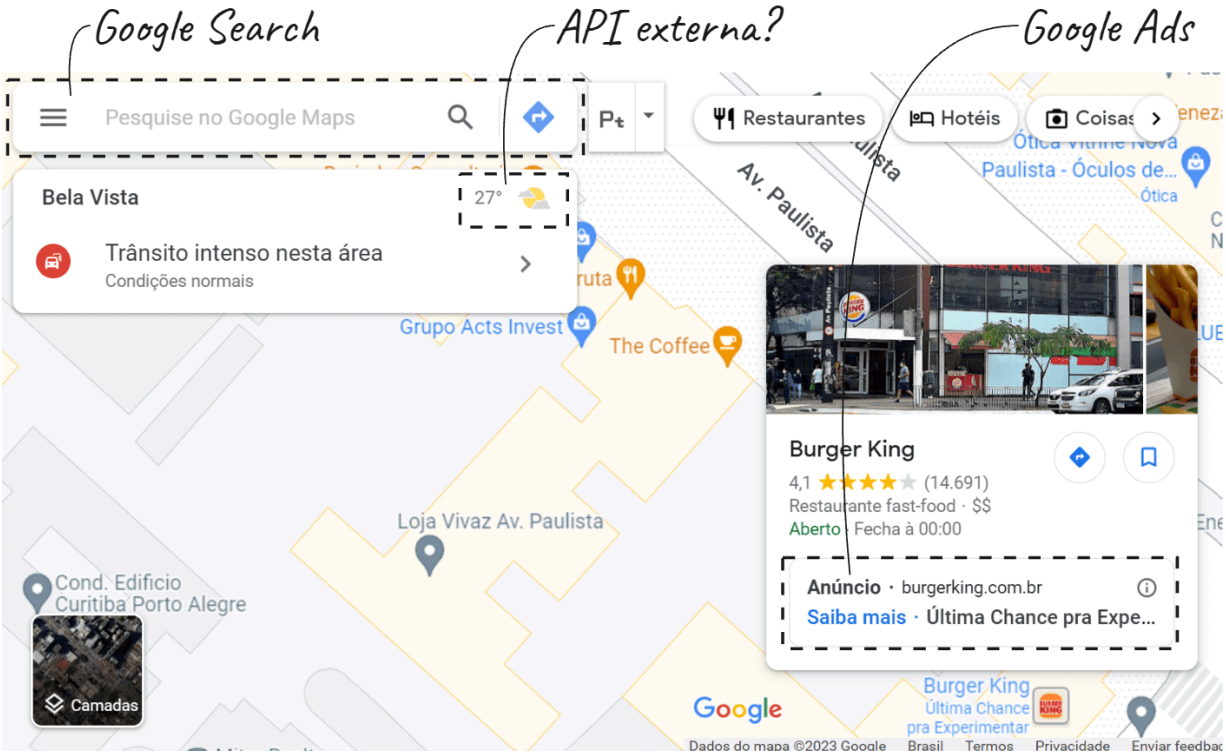


Figura 8.1: APIs que o Google Maps utiliza.

E não para por aí. Os dados do mapa pertencem à plataforma Google Maps e podem ser reutilizados por produtos do próprio Google e por outras empresas. A exemplo do iFood, a pessoa que faz um pedido consegue acompanhar a localização do entregador no mapa. Este é um caso típico de adição de funcionalidade por meio de reutilização. O time do app não precisou desenvolver a própria solução, o que levou à economia de tempo para disponibilizar esta funcionalidade aos usuários.

Comparando com produtos físicos, o potencial da combinação de APIs é infinito e além. Enquanto, sobre uma plataforma de carros, cria-se mais do mesmo (carros, apesar de modelos diferentes),

plataformas digitais habilitam **inovação**. Acima delas, é possível criar modelos de negócios que não existiam até então, mostrando que o valor de uma plataforma pode ser realmente poderoso! Assim foi com a Uber, que criou um modelo de negócio inteiramente novo sobre a plataforma Google Maps. Se não utilizassem algo existente, talvez a expansão para outros países ainda seria um sonho distante.

Esses exemplos mostram que a reutilização **acelera o *time-to-market***. Além de ser uma baita **vantagem competitiva**, ao acelerar a entrada no mercado, ela **antecipa a geração de receita**.

Novas fontes de receita

Plataformas internas não geram receita diretamente, ainda assim atuam na maneira como a empresa ganha dinheiro. Em outras palavras, APIs proporcionam a capacidade de produtos derivados da plataforma se tornarem novas fontes de receita. Então, supondo que cada solução derivada seja um fluxo de receita, a plataforma se torna provedora de múltiplos fluxos de receita.

A empresa também pode decidir disponibilizar as APIs no mercado, isto é, para clientes externos, liberando o potencial de se tornar mais um fluxo de geração de receita. Assim fizeram o Google Maps, a Amazon e até a Volkswagen.

Mais produtividade a menor custo

As empresas podem desenvolver uma família de produtos com base em uma plataforma comum, em vez de começarem do zero todas as vezes. Por exemplo, o iFood possui três perfis de usuários e há um app específico para cada um deles — ou seja, três apps. Mas tem clientes que usam Android, outros usam iPhone e há quem prefira usar a versão desktop no navegador (disponível para restaurantes e consumidores), totalizando oito produtos — não tem versão desktop para entregadores. Todos eles usam a Order Platform para acionar operações dentro do iFood e exibir o status do pedido aos três perfis

de usuários. Se tivessem que replicar os recursos, seria um grande desperdício de tempo e esforço.

Como o recurso está pronto para uso, os times que usam a plataforma são mais produtivos, o que reflete em **ganho de eficiência** e **redução de custo**. E mais, a arquitetura de microsserviços melhora a produtividade do próprio time que trabalha na plataforma e reverbera por todo o negócio até chegar aos usuários finais.

Para ilustrar, temos como exemplo a Atlassian, que encontrou dificuldade para escalar o Confluence e migrou de monolito para microsserviços. No site da Atlassian, o autor do post *Microservices vs. monolithic architecture* (HARRIS, c2022) afirma que, além de autonomia para os times, essa mudança aumentou a frequência de deploy de uma vez por semana para duas ou três vezes por dia.

A junção de APIs, microsserviços e tecnologia em nuvem trouxe resultado ainda mais expressivo para a Magazine Luiza. De acordo com o e-book *The API Product Mindset*, isso alavancou de 8 deploys por mês para mais de 40 por dia. Com esse ganho de agilidade "*a empresa conseguiu reformular a plataforma dos seus apps mobile e site de comércio eletrônico e introduzir dezenas de produtos internos*" (GOOGLE CLOUD, 2018; tradução livre).

Além da eficiência dos times, plataformas bem elaboradas criam valor ao garantir o uso eficiente dos recursos. Além de evitar processamentos desnecessários, o custo por unidade diminui à medida que o volume de transações aumenta — assim como o preço de atacado. Se quanto maior o uso da plataforma, menor o custo por transação, isso significa que **a margem de lucro aumenta**.

Os requisitos não funcionais de escalabilidade, alta performance e alta disponibilidade impactam diretamente o usuário final e levam, por exemplo, a evitar perda de conversão de vendas, o que costuma estar diretamente atrelado à **rentabilidade**.

Redução de riscos

Nos casos em que determinado componente é reutilizado em 10 aplicações, por exemplo, ao apresentar um problema, este se espalha pelo negócio e se torna um problema muito maior. Para contrapor a possibilidade de danos, é natural que a qualidade seja privilegiada, o que resulta em mais um benefício: **redução de risco**.

Assegurar que o código não tenha bugs toma mais tempo, mas é provável que leve menos tempo do que se tivesse que garantir qualidade em 10 aplicações separadas. Uma vez que apps/produtos que derivam da plataforma herdam a qualidade, o resultado é uma linha de **produtos de qualidade superior**.

Além do design da arquitetura, a escolha de integração com os serviços complementares também define o cumprimento do valor.

8.3 Tecnologia em nuvem agrega valor

Uma empresa em rápido crescimento espera aumento significativo no volume de transações. Para fornecer uma solução abrangente, aproveitar os recursos da plataforma de nuvem, como escalabilidade, flexibilidade, alta performance, segurança e privacidade, se mostra uma boa escolha.

A nuvem oferece mais flexibilidade em comparação com a hospedagem em um servidor local (*on-premise*). Quando o sistema precisa de banda extra, um serviço baseado em nuvem pode atender a essa demanda instantaneamente, em vez de passar por uma atualização complexa (e cara) de infraestrutura de TI. Isso faz diferença na eficiência da organização.

Os serviços baseados em nuvem fornecem recuperação rápida para todos os tipos de cenários de emergência (*disaster recovery*), desde desastres naturais até falta de energia. Segundo a Salesforce (c2023), "*enquanto 20% dos usuários da nuvem afirmam atingir recuperação de desastres em quatro horas ou menos, apenas 9% dos usuários fora da nuvem podem afirmar o mesmo*" (tradução livre).

A nuvem garante segurança e privacidade e, de acordo com um artigo da SAP Insights (c2023), segurança é fator decisivo na adoção da nuvem, não só para proteger os negócios de possíveis ataques hacker, mas também para garantir acesso apropriado apenas aos funcionários, clientes e parceiros, mantendo-se em "conformidade com as leis locais" (tradução livre).

8.4 Casos de uso específicos podem ser o diferencial

A pesquisa *The State of API Integration 2021* (CLOUD ELEMENTS, 2021), com respostas de quase 250 entrevistados de 30 países, mostra que a maior demanda que os respondentes recebem de clientes e parceiros em relação à integração da API é, em primeiro lugar, de "*APIs que atendam a uma necessidade de negócio específica (42%), seguida por melhor documentação (27%)*" (tradução livre).

Plataformas de mercado jamais conseguirão atender a casos de uso específicos do negócio. É aqui que reside a grande oportunidade, o diferencial, o maior valor que uma plataforma interna pode fornecer.

Para o desenvolvimento da nova Order Platform, foi considerado usar uma solução de mercado para a máquina de estados (*state*

machine). Foram avaliadas algumas alternativas e nenhuma delas era aderente às necessidades de acompanhamento de status de refeição ou de mercado, nem à capacidade de customização.

No *discovery* descobrimos que, além de máquinas de estado diferentes para cada unidade de negócio, eram necessárias máquinas diferentes para a mesma unidade. Por exemplo, se o usuário escolhe retirar o pedido no restaurante, não faz sentido ter os status de entrega. O consumidor precisa saber quando o pedido está pronto para retirar. Nenhuma solução de mercado habilitava tamanha versatilidade.

Uma boa arquitetura garante que as capacidades proporcionadas pela plataforma sejam como peças de montar (*building blocks*), de modo que devs usuários sejam capazes de combinar recursos de maneira rápida e fácil, como fariam com peças de Lego™; indo além, a expectativa é que isso **destrave casos de uso** que possam surgir, ou seja, que a plataforma tenha a capacidade de se expandir para suportar casos de uso adicionais.

Cuidado com customizações

Como plataformas atendem a múltiplos casos de uso, é comum as pessoas usarem o argumento de que precisam ser genéricas. É verdade, o ideal é que seja genérica, mas é preciso cuidado para não deixar a plataforma genérica demais a ponto de não atender às necessidades da empresa, tornando-a inútil.

O oposto também é verdadeiro. Customizar para um caso de uso pode comprometer a capacidade de atender aos demais contextos. O ideal é que ela seja configurável, assim a base genérica é preservada e configurações adaptam aos variados casos de uso. O equilíbrio vem da compreensão de quais funcionalidades ou processos se devem padronizar para ela ser mantida como base comum a todos os apps e onde a variedade é necessária.

Se plataformas têm potencial de fornecer benefícios de produtividade, rápido crescimento em participação de mercado e receita, ao mesmo tempo em que cortam custos, habilitam a inovação e a entrega rápida de novos produtos, por que há stakeholders que não veem esses benefícios?

8.5 Por que stakeholders não veem valor?

Se mesmo para PMs que estão expostos a termos técnicos no dia a dia alguns assuntos deles são difíceis, imagine para quem não lida com tecnologia de perto. É natural que algumas pessoas não entendam completamente o que são plataformas e como elas podem melhorar a eficiência e a escalabilidade da empresa. Mas alguns acontecimentos contribuem para isso.

Design da arquitetura projetada para curto prazo

Um deles é o que vimos até agora: design da arquitetura. Arquitetura mal elaborada compromete o pleno potencial das *features*, colocando os valores em risco.

Como já dito, é comum a plataformização iniciar sem planejamento, apenas como integração entre sistemas, com apenas um caso de uso em mente. Então, quando alguém precisa de alguma coisa diferente do que a plataforma oferece, a resposta do time da plataforma é "não dá", afinal ela não foi originalmente projetada para ser configurável a novos casos de uso. Nesta situação, o time que se tornaria cliente da plataforma tem que desenvolver a própria solução.

Via de regra, é esperado que plataformas sejam *enablers* (facilitadoras, que tornam possível), e quando falamos "não dá",

estamos sendo *disablers* (incapacitantes, que impedem). É claro que os stakeholders vão questionar o valor da plataforma!

Plataforma boa passa despercebida

Infelizmente, o valor de artefatos estruturais fica evidente por motivos negativos. Quanto mais qualidade tem a plataforma, menos ela é lembrada. Soluções estruturais têm essa característica, como a energia elétrica: só percebemos o quanto faz diferença quando falta. Plataforma boa, excelente, sensacional implica em não ser notada na empresa e, por conta disso, as pessoas podem não perceber o valor que ela tem (até que ela caia).

Por isso, quando chega o momento de rever orçamentos e planejar, a alta gestão pode questionar: para que essa plataforma existe se não vejo acontecer nada?

Quando construímos uma solução competente, a empresa opera melhor; contudo, os stakeholders não percebem que a plataforma é a propulsora da harmonia. Neste caso, a plataforma passar despercebida significa que o cliente está satisfeito e nem sabe.

Costume com resultado de curto prazo

Lembra que vimos no capítulo *Problemas e soluções* que é comum negócios iniciarem com produtos baseados em monolitos? Monolitos são mais rápidos de construir, o que deixa os stakeholders acostumados a observarem resultados no curto prazo e com experimentos rápidos. Os stakeholders mantêm a expectativa de resultados imediatos. Acontece que plataforma é um plano para longo prazo. Eles não entendem a perda de velocidade no começo da construção de uma plataforma ou quando há novas implementações grandes.

Melhorias na estrutura da plataforma não são fáceis de prototipar e de testar rápido. Muitas vezes, também é difícil demonstrar causa e

efeito (como o recurso X levou ao impacto Y). Conectar a uma meta de negócio pode ser difícil, porque plataformas são projetadas tentando prever o que vai acontecer daqui a alguns anos. Temos que inferir coisas como: o negócio vai crescer 10 vezes, então tem que suportar 10 vezes mais tráfego? Talvez 100 vezes mais? Independente do número, somente no futuro será observável.

Isso não significa que não seja possível ver retornos em curto prazo. Os conceitos das metodologias ágeis se aplicam para plataformas também, por exemplo, entregando um caso de uso por vez. Mas a fase inicial, a fundação, pode levar meses para ser construída.

Para agravar a situação, os investimentos costumam ser altos no início e o retorno vem ao longo do tempo, conforme a empresa cria mais apps, por isso, o retorno pode parecer lento.

Assim, mesmo que a promessa da proposta de valor seja entregar todos os benefícios possíveis e que os stakeholders conheçam todos eles muito bem, a percepção de valor é comprometida devido ao tempo que leva para se construir plataformas. Tempo é um custo alto para empresas que visam crescimento “agora”.

Plataforma desconectada dos objetivos de negócio

Às vezes, a construção da plataforma está num momento tão diferente do que está planejado no curto prazo que até o próprio time da plataforma não vê valor.

Quando entrei para o time da Order Platform, percebi que o time não entendia por que estava criando uma nova versão da plataforma. No ponto de vista do pessoal, a nova plataforma era meramente uma mudança de tecnologia e o time não via necessidade dessa mudança. Minha dúvida era: a demanda para construir a nova plataforma entrou sem que o time recebesse informações ou ela era potente demais para o que o iFood precisava?

Era uma mistura de ambos. E pior, as poucas informações que tinham estavam desalinhadas. No *discovery*, entendemos melhor aonde o iFood pretendia chegar e quando. Entendemos o que não estava bom para os clientes finais e descobrimos que uma funcionalidade do app, importantíssima para os clientes e para a operação, era subutilizada devido à falta de recursos da Order Platform.

Depois que juntei as peças e apresentei para o time que (1) a mudança de tecnologia vislumbrava o futuro crescimento do iFood, que (2) a máquina de estados atual gerava problemas e "gambiarras" em várias partes da empresa e (3) o primeiro caso de uso a ser destravado — na situação, a funcionalidade subutilizada —, a percepção de valor que o time tinha mudou completamente, inclusive a da alta gestão.

Infelizmente, algumas plataformas resolvem problemas que não refletem diretamente no negócio. Elas resolvem problemas de TI, como expor tabelas de banco de dados, automatizar processos internos ou melhorar a arquitetura do sistema, importantes de se resolver por motivos técnicos, não por motivos de negócios. Mesmo sem grande impacto nas operações do negócio, sem movimentar o ponteiro com relação à meta de vendas ou de produtos, são problemas importantes a serem resolvidos, porém difíceis de se mostrar valor. Muitas vezes, esse é um caso que deve ser apoiado e "vendido" dentro da empresa pelo CTO.

Esses desalinhamentos exigem mudanças nas formas de trabalhar e pensar, o que leva tempo para incorporar como o "novo normal" — *desculpe o termo desgastado pós-pandemia*. Por isso, precisamos evangelizar, endereçando o item que vimos no começo do capítulo: *negligência ao articular o valor do compartilhamento*.

Antes, vamos relembrar o significado de valor.

8.6 O que é valor e o que é relevante comunicar

Valor é a **percepção** dos clientes e usuários sobre o produto. Para concluir se algo tem valor, as pessoas ponderam os benefícios que elas obtêm do produto e os custos que assumem para o consumo, a conhecida balança *custo-benefício*.

Os custos para o consumidor podem ser monetários ou podem ser de tempo, de esforço, de risco, de oportunidade; e os benefícios percebidos residem na avaliação da qualidade de três aspectos, como explica o livro *Gestão e desenvolvimento de produtos e marcas* (IRIGARAY et. al, 2011):

- **Qualidade do produto**: características, garantia, durabilidade, desempenho, facilidade de uso;
- **Qualidade dos serviços**: empatia no atendimento, transmissão de confiança;
- **Qualidade da experiência**: atmosfera do ambiente, decoração da loja, propaganda, publicidade, promoção.

Para plataformas, esses itens se traduzem em:

- **Características**: interface síncrona, interface assíncrona, estrutura dos dados;
- **Garantia**: recuperação de desastres, tolerância a falhas;
- **Durabilidade**: atualizações da plataforma;
- **Performance**: baixa latência;
- **Facilidade de uso**: APIs documentadas, fáceis de entender e usar;
- **Empatia no atendimento**: agilidade no suporte aos times quando precisarem;
- **Transmissão de confiança**: explicação do porquê das coisas.

Em relação à qualidade da experiência devemos nos atentar ao processo para adoção e ferramental para os devs usuários, por exemplo, ambiente para testes.

Podemos nos valer desses fatores ao definir as soluções e elevar a percepção de valor, "*aumentando os benefícios econômicos, funcionais ou emocionais e/ou a redução de um ou mais tipos de custo*" (KOTLER; KELLER, 2018).

Claro que depende de a quem a mensagem se dirige. Em ambientes corporativos, apelos como "ganhar mais dinheiro" e "diminuir custos" costumam ser de grande valor — no capítulo *Estrutura de custos e fluxos de receita* veremos mais sobre isso. Um benefício emocional talvez não faça tanta diferença para alguém da alta gestão se comparado ao benefício econômico. Mas se o *budget* for infinito, parecer mais inteligente ou competente pode ser relevante para satisfação pessoal.

Sobre como comunicar o valor, a regra de marketing e vendas (podemos incluir gestão de produtos) é clara:

1. A mensagem deve ser elaborada com base na audiência à qual a comunicação se destina, porque o que é relevante para uma pode não ser relevante para a outra. Isso significa que se pode mencionar uma coisa ao cliente e outra ao usuário, por exemplo.
2. O que se "vende" são benefícios em vez de *features*, porque as pessoas não compram produtos, elas compram benefícios. Elas não se importam *como* o valor é criado, elas só querem saber *qual* é o valor, o que vai melhorar na vida delas.

Será que essa regra é válida para comunicar valor de plataformas? Se sim, a capacidade da plataforma de escalar de acordo com o crescimento da empresa é *feature* ou benefício? E quanto à qualidade, isenção de bugs, resiliência? E facilidade de uso das APIs? Às vezes as coisas podem se confundir. Então, vejamos a diferença entre *features* e benefícios.

Features

Features são fatos sobre o produto, as especificações técnicas que explicam o que ele faz e como funciona. Mencionar as *features* adiciona credibilidade e ajuda a tangibilizar o que esperar do produto, inclusive no quesito experiência. Por exemplo, uma caneta: tinta azul, ponta esferográfica, corpo colorido e emborrachado. Estas são simplesmente as especificações que fazem da caneta o que ela é.

E o que faz a plataforma ser o que é? Infraestrutura, design da arquitetura e das APIs definem como a plataforma funciona e o que ela faz: processa pagamentos, transmite mensagem, bloqueia acesso etc. Dependendo das escolhas, uma plataforma pode ser (ou não):

- Escalável: acompanha o crescimento da demanda, como picos e quedas de tráfego;
- Flexível: não afeta o sistema como um todo quando há atualizações e mudanças;
- Resiliente: continua funcionando mesmo quando há interrupções por erros ou falhas e se recupera rapidamente;
- Robusta: suporta erros e condições adversas sem falhar ou interromper o funcionamento;
- Segura: possui políticas de segurança para lidar com privacidade e controle de acesso.

Benefícios

Os benefícios explicam o valor gerado pelas *features*, qual resultado os clientes obterão que demonstra que vai melhorar a vida deles (ou da empresa). Benefícios dão um motivo para os clientes usarem. No exemplo da caneta, os benefícios relacionados às *features* citadas poderiam ser: experiência de escrita suave e agradável (ponta esferográfica e corpo emborrachado), diversas cores para combinar com os itens da sua mesa (corpo colorido).

Já vimos bastante sobre os benefícios de plataformizar neste capítulo, mas, só para exemplificar, vou citar dois deles e os seus

respectivos valores para o negócio:

- Escalabilidade: habilita o crescimento do negócio e, como consome somente os recursos necessários para atender à demanda, tem o melhor custo-benefício;
- Disponibilidade: garantia de que os usuários conseguirão acessar o app a qualquer momento e em qualquer tipo de dispositivo, evitando perda de conversão de vendas.

Mensagem certa para a audiência certa

A maior parte da nossa audiência são as pessoas de engenharia. Para elas, a *feature*, por si só, indica o benefício. Já para stakeholders de negócios, *feature* pode não significar nada, a não ser que seja traduzido em um benefício que demonstre ajudar o negócio a ganhar mais dinheiro, economizar, diminuir riscos e acelerar o crescimento. Isso mostra que, dependendo do público, ela pode (ou até deve) mencionar os aspectos técnicos.

Então, para pessoas de negócios, poderíamos dizer: "a plataforma suporta picos de tráfego 100 vezes maior do que o volume regular, sem perda de performance, então não corre o risco de perder conversão de vendas na próxima Black Friday".

Já para as pessoas de engenharia, poderia ser simplesmente: "a plataforma escala horizontalmente de acordo com a demanda". Escala horizontal transmite um benefício preciso: aumenta os recursos se o tráfego aumentar. Isso é suficiente para elas confiarem que é baixo o risco de a funcionalidade que será adicionada no app ficar lenta ou fora do ar.

Entretanto, nem todo dev estudou engenharia ou ciência da computação. Para este público, talvez um mix entre as duas frases seja o ideal: "a plataforma escala para processar volume de dados até 100 vezes maior, sem perda de performance". Devs usuários não se importam se por trás da API roda um monolito ou se a

arquitetura é de microsserviços, mas isso pode fazer diferença para a alta gestão de tecnologia, porque ela tem a visão de como isso impacta o todo da organização.

Para devs, faz diferença a linguagem de programação da API, porque talvez a integração seja inviável. Também pode ser uma informação importante para a alta gestão, porque, se eles sabem que a maior parte dos apps da empresa usa a linguagem A, e a plataforma só suporta a B, provavelmente vão pedir para que ela suporte a A.

Independente do perfil do público, há sempre o mínimo a se esperar de resultado. Quando alguém considera comprar uma caneta (barata ou cara), o mínimo que se espera é que flua tinta sobre o papel ao escrever. Qualquer stakeholder também tem expectativas sobre uma plataforma: no mínimo, que ela seja confiável, estável e previsível, porque a empresa depende das plataformas para que a operação do negócio flua.

Fluir bem depende das escolhas em relação ao tipo de tecnologia, como ela é implementada (arquitetura do software) e a facilidade de uso (design da API).

8.7 Juntando tudo

Na prática, ninguém quer plataformas ou APIs: os times querem desenvolver rápido o app deles; pessoas de negócios querem que este app que os times desenvolveram gere resultado rápido para o negócio. Plataformas são o meio para se chegar lá.

Alegar que a plataforma fornece tais benefícios não basta. A proposta de valor, que muitas vezes é intangível, precisa ser materializada — e aqui pode estar mais uma razão pela qual alguns stakeholders não veem valor. Se você diz que a plataforma vai acelerar o *time-to-market* de novos produtos e a API é difícil de usar,

tomando muito tempo para integrar, isso mostra que o custo para consumo está alto, portanto, a percepção de valor cai.

As empresas que participaram da pesquisa *State of API Integration 2021* (CLOUD ELEMENTS, 2021) e conseguiram materializar a aposta em plataformas afirmam que obtiveram os seguintes benefícios de integração via API: aumento da velocidade para atender às demandas de negócios (44%); aumento de inovação (40%); aumento de produtividade (38%); diminuição de custos (23%); e aumento de receita (18%), entre outros.

Qual empresa não quer esses benefícios? Uma das maneiras de conquistá-los é por meio da reutilização dos recursos. Plataformizar os recursos vem acompanhado de desafios estratégicos, financeiros e de gestão. A execução normalmente leva vários ciclos, e é difícil equilibrar as necessidades de curto prazo e de algo maior, de longo prazo.

Isso coloca alguns benefícios da plataformização em xeque. Enquanto o desenvolvimento do app que usa a plataforma se torna mais rápido, o desenvolvimento da plataforma é mais lento. Visto que qualquer fragilidade da plataforma será herdada por todos os produtos que derivarem dela, são necessárias garantias de funcionamento.

Enquanto produtos obtêm rentabilidade direta quando os clientes compram, até que as plataformas sejam amplamente usadas, elas são vistas como investimento, cujos custos, no fim das contas, são compartilhados com os apps que derivam da plataforma. Então, fica a pergunta: reduz o custo, de fato? Difícil dizer.

Contudo, uma coisa se mostra pertinente: acelera o *time-to-market* de novos produtos e funcionalidades, o que antecipa geração de receita. Colocar um app no mercado antes dos concorrentes ou lançá-lo antes dos seus clientes mudarem para o concorrente parece o melhor dos mundos. Isso mostra que a estratégia de plataforma atua como alavanca de **vantagem competitiva**, sim!

Então, para fechar o capítulo, vamos preencher o Lean Canvas. É importante lembrar que o conteúdo do quadrante *Proposta de Valor* deve ser claro para o time. Então, em princípio, pode ser preenchido com termos técnicos, juntando *features* e respectivos benefícios. É melhor reservar as "mensagens de efeito” para as apresentações.

Os casos de uso podem ser uma descrição técnica detalhada das interações entre um sistema e os usuários, mas para demonstrar valor, é preferível colocar o que é possível se fazer. Na imagem, vemos um exemplo que poderia ser útil a qualquer *e-commerce*.

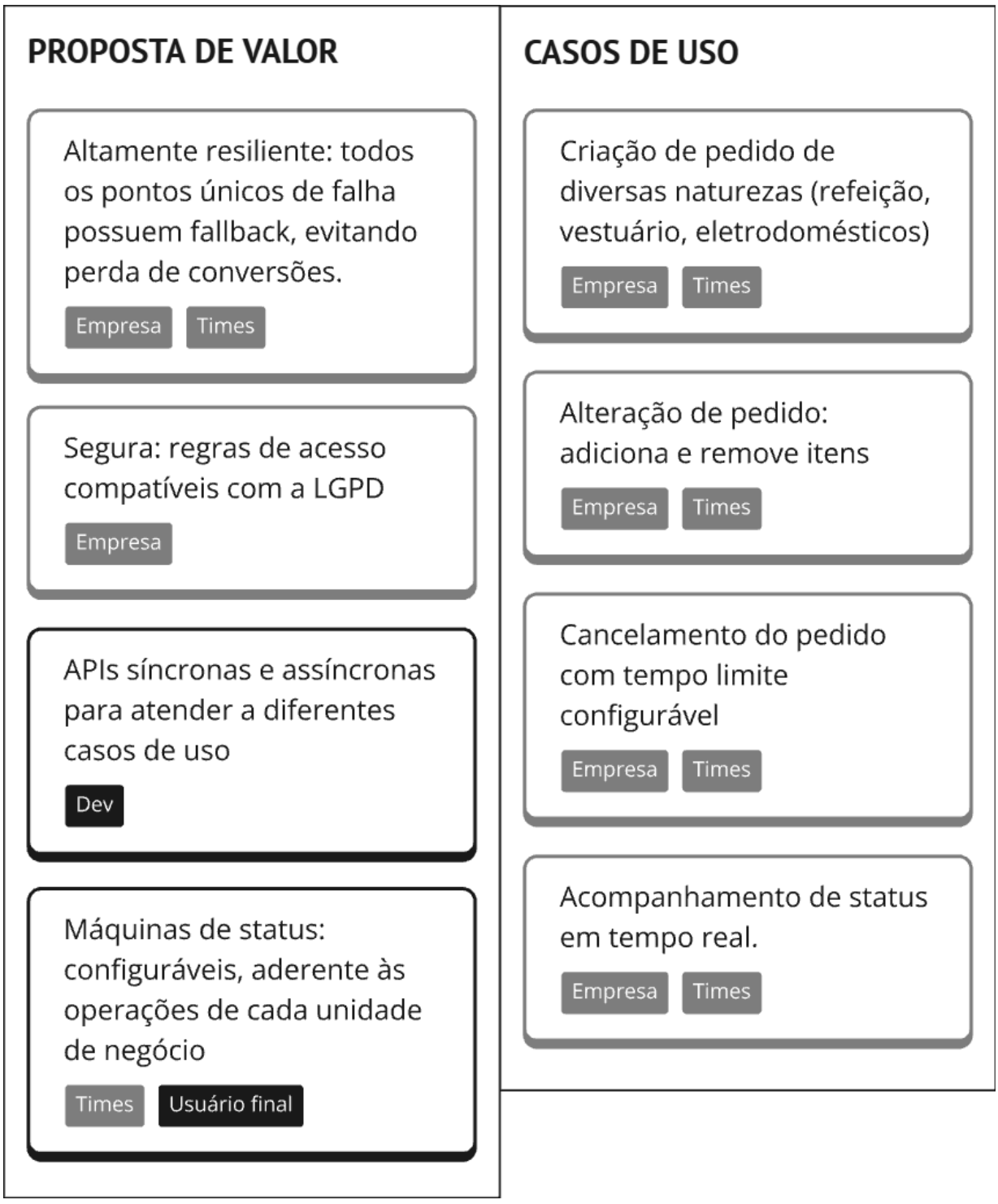


Figura 8.2: Lean Canvas: Proposta de Valor e Casos de Uso.

É uma boa fazer *benchmark* para conhecer os casos de uso que as plataformas de mercado habilitam e como comunicam o valor.

Vejamos como exemplo o Twilio, uma das plataformas mais elogiadas quando o assunto é experiência do dev. Repare que a mensagem é direcionada a pessoas técnicas usando os termos "escaláveis" e "linhas de código".

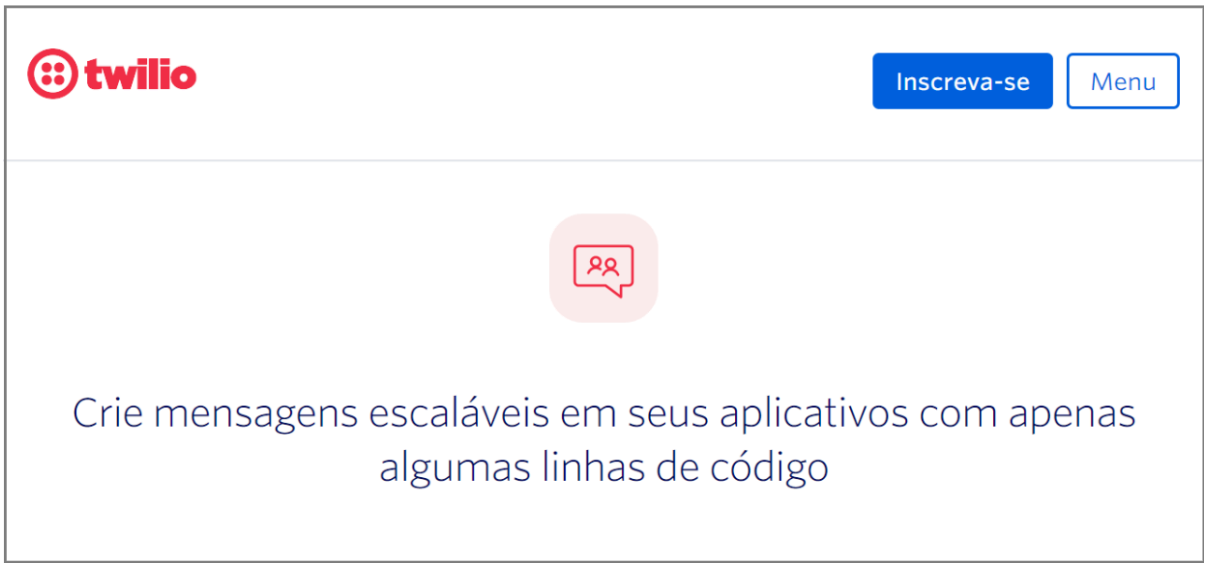


Figura 8.3: Twilio. (https://www.twilio.com/pt-br/messaging/programmable-messaging-api)

Na seção de casos de uso do Twilio, o site mostra ideias de cenários em que as notificações são úteis. Você pode fazer o mesmo para ajudar a tangibilizar a utilidade das funcionalidades na mente dos stakeholders.

Figura 8.4: Twilio. (https://www.twilio.com/pt-br/messaging/programmable-messaging-api)

Como vamos saber se as escolhas estão indo em direção ao que foi prometido na proposta de valor? Precisamos definir e acompanhar algumas métricas. É o assunto do próximo capítulo.

Capítulo 9
Métricas

Seja qual for a proposta de valor, é mais fácil conseguir *buy-in* dos stakeholders e mostrar valor quando se tem números, o que torna o quadrante *Métricas* do *Lean Canvas* um tanto polêmico na gestão de plataformas. Isso porque algumas vezes não é possível relacionar a construção de plataformas com métricas de negócio. A situação se agrava quando envolve coisas que são imensuráveis. Por exemplo, como se mede o potencial de entrada em novos nichos em tempo recorde? E, pior, como mostrar numericamente esse potencial para os próximos cinco anos?

As empresas ainda estão aprendendo como lidar com plataformas, então é natural que busquem padrões conhecidos e os apliquem no desconhecido. O problema é que as pessoas tendem a atribuir métricas que não correspondem ao que está sob o controle do time da plataforma. Usam a mesma régua para medir coisas de escalas e naturezas diferentes. É como querer explicar fenômenos do nível subatômico usando física clássica. Se alguém quer entender o que acontece no nível subatômico, precisa se valer das leis da física quântica. É esta a mudança que precisamos fazer: medir o que é de competência de plataformas — normalmente, coisas diferentes de produtos *customer-facing*. E que coisas são essas? As **métricas de sucesso** e **métricas de saúde** da plataforma.

As *métricas de sucesso* são correlatas às de produtos, como já conhecemos. Então, para efeito didático, vou usar como guia um framework trivial para PMs, a *Pirate Metrics* (AARRR), que, para produto SaaS (*software as a service*), teria o seguinte funil e definições para cada fase:

- Aquisição (*acquisition*): os usuários se cadastram para testar o produto;
- Ativação (*activation*): os usuários usam/testam;

- Retenção (*retention*): os usuários usam regularmente;
- Receita (*revenue*): os usuários pagam para continuar usando;
- Recomendação (*referral*): os usuários recomendam o produto à outras pessoas.

Figura 9.1: Funil AARRR de produto SaaS.

Agora veja como o funil AARRR se traduz para as métricas de plataforma — é comum as ferramentas de monitoramento usarem o nome das métricas em inglês, por isso, assim foram mantidas.

9.1 Aquisição

A fase de *aquisição* de um produto SaaS se refere ao cadastro de usuários para testar o produto. Da mesma forma, é importante

saber quem são os usuários das APIs, que, neste caso, são os apps conectados à plataforma.

O ideal é que os times que vão usá-la tenham um cadastro para gerar algum tipo de identificador, por exemplo, *API key*. Esse cadastro pode ser por meio de contato direto com o time da plataforma, um formulário no Slack ou no portal do desenvolvedor, por exemplo. Para efeito de identificação, basta o nome da aplicação que vai "plugar" na API e é útil saber também a *squad* responsável, para facilitar a comunicação entre times.

O que importa aqui é tornar possível medir a quantidade de aplicações pelas quais a API é reutilizada, e o número de *API keys ativas* provê dados para a métrica *unique API consumers* (consumidores únicos da API) — apps que consomem a API, que pode ser encontrada sob a nomenclatura *active apps* (apps ativos).

Note que a quantidade de times ou devs usuários é indiferente — afinal, um time pode ter três aplicações que usam a API, assim como cada app pode ser desenvolvido por cinco devs. Isso mostra que o número de times ou devs usuários não representa a reutilização.

A criação da *API key* indica que o time de determinado app pretende adotar a solução. Por ser uma plataforma interna, dificilmente uma *squad* vai testar a API apenas por curiosidade. Então, se a *API key* for criada e o dev usuário não avançar no funil depois disso, vale a pena entender com o time do app se há algum atrito no qual o time da plataforma possa atuar e, assim, adquirir o cliente interno.

A quantidade total de *API keys* costuma ser irrelevante. Pense, o que você pode fazer se há 10 chaves cadastradas e 5 não são ativas? Pode ser que alguns apps deixaram de existir, portanto, não há uma ação para recuperar ex-clientes. No fim das contas, isso gera mais ruído do que informação. No entanto, é uma boa atentar-se à

geração de novas chaves, para que, como já dito, seja possível agir e evitar que o potencial time cliente desista da integração.

Para saber o potencial de aquisição, precisamos conhecer nosso mercado interno. Para isso, podemos usar a abordagem TAM, SAM e SOM.

TAM, SAM e SOM

Tradicionalmente, o tamanho total do mercado disponível (*Total Available Market* — TAM) refere-se ao máximo de receita que uma empresa pode gerar de vendas de determinado produto em um mercado específico, se não houvesse concorrência. No contexto de plataforma, poderíamos considerar todos os apps da empresa. Mas, para evitar falsas expectativas e cobranças desnecessárias, o ideal é considerar apenas os apps que têm casos de uso aos quais a plataforma é capaz de atender.

Vamos imaginar uma plataforma que processa pagamentos de cartão de crédito em uma empresa que possui 100 aplicações, e 5 delas possuem algum tipo de venda e precisam receber pagamentos. Supondo que cada app vende R$ 1 milhão por ano, o TAM seria R$ 5 milhões. Como o objetivo de plataformas é a reutilização, em vez de considerar o volume monetário, podemos definir o potencial com base em usuários únicos; assim, nesse exemplo, o TAM é 5 apps.

O mercado alcançável e com potencial de ser atendido (*Serviceable Available Market* — SAM) é o volume total vendido por todos os *players* de um segmento específico, o que leva ao tamanho do mercado-alvo. Continuando o exemplo, vamos supor que quatro apps aceitem cartão de crédito e um deles só aceite boleto. O SAM seria 4 apps, ou R$ 4 milhões.

Já a parte do mercado que uma empresa pode atender (*Serviceable Obtainable Market* — SOM) é a fatia do mercado que o produto tem condições de capturar. Em tese, a plataforma de pagamentos tem

condições de atender aos quatro apps, porém é mais útil considerar somente os apps que já a utilizam, para separar do quanto ainda é possível capturar. Vamos supor que, no exemplo apresentado, 2 apps usam a plataforma e 2 usam plataformas de mercado, portanto, o SOM seria 2 ou R$ 2 milhões.

Esse é o método para quem tem apego a frameworks. Agora, para quem prefere praticidade, sem rodeios: se a empresa tem cinco apps com canal de vendas e, dentre esses, dois utilizam a plataforma e um deles não é *target*, só há dois para aquisição pelo time da plataforma. Simples assim! E o que você pode fazer para expandir seu mercado? Que tal diversificação de ofertas? Se você passar a processar pagamentos por boleto, é a oportunidade para conquistar o cliente remanescente e também oferecer aos clientes que você já tem.

Há ainda uma forma mais pragmática de pensar. O objetivo da plataforma pode ser a reutilização por 100% dos apps da empresa que são canais de vendas, simplesmente porque os milhões de reais que transaciona são irrelevantes quando o que se quer é que todos, desde o produto vaca leiteira até o menor experimento, usem a sua plataforma. Porém, para que as vendas sejam bem-sucedidas, o time da plataforma precisa garantir que sejam processadas 100% das transações, inclusive do pequeno experimento de algum time cliente.

E quanto ao potencial de futuras aquisições? É impossível mensurar, pois depende da estratégia da empresa. Por exemplo, para a plataforma de pagamentos do nosso exemplo, vai depender de lançamento de mais apps que funcionem como canais de vendas.

9.2 Ativação

Cadastro de usuários não significa grandes coisas, pois só é possível perceber valor quando se usa o produto. As métricas que geralmente demonstram *ativação* são relacionadas ao uso de determinadas funcionalidades, expostas no início da jornada, de modo que o usuário chegue o quanto antes ao famoso *aha moment* — a solução para o problema se torna clara.

O *aha moment* de APIs ocorre quando o dev faz a primeira requisição (*request*) à API e obtém "sucesso" de resposta. O tempo decorrido desde o cadastro até a primeira requisição de sucesso via API é conhecido como *time to hello world* (TTHW).

Time to hello world **(TTHW)**

O TTHW ajuda a entender a experiência do dev (DX) — cadastro, documentação, tutoriais. Uma API difícil de entender demora mais tempo para se começar a usar e, se a proposta de valor está atrelada a acelerar o *time-to-market*, é preciso garantir que o *aha moment* se dê o quanto antes. Essa métrica mostra se há algo a ser melhorado na jornada que reduza o atrito e o esforço do dev para chegar ao "*hello, world*".

Algumas pessoas calculam o TTHW a partir da requisição (que resulta em sucesso) feita do ambiente de produção. Outras consideram como primeira, a requisição vinda do ambiente de desenvolvimento ou teste via Postman. Neste caso, atribui-se outra medida de tempo quando ocorre a primeira requisição de sucesso feita do ambiente de produção, métrica nomeada de *time to value* (tempo para obter valor).

Separar o *onboarding* em dois estágios — *time to hello world* e *time to value* — torna claro o momento em que ocorrem dificuldades que seriam motivos para desistir do uso (*drop-off*), se fosse uma plataforma externa. É o que mostra o gráfico a seguir da plataforma Moesif API Analytics (GILLING, 2022):

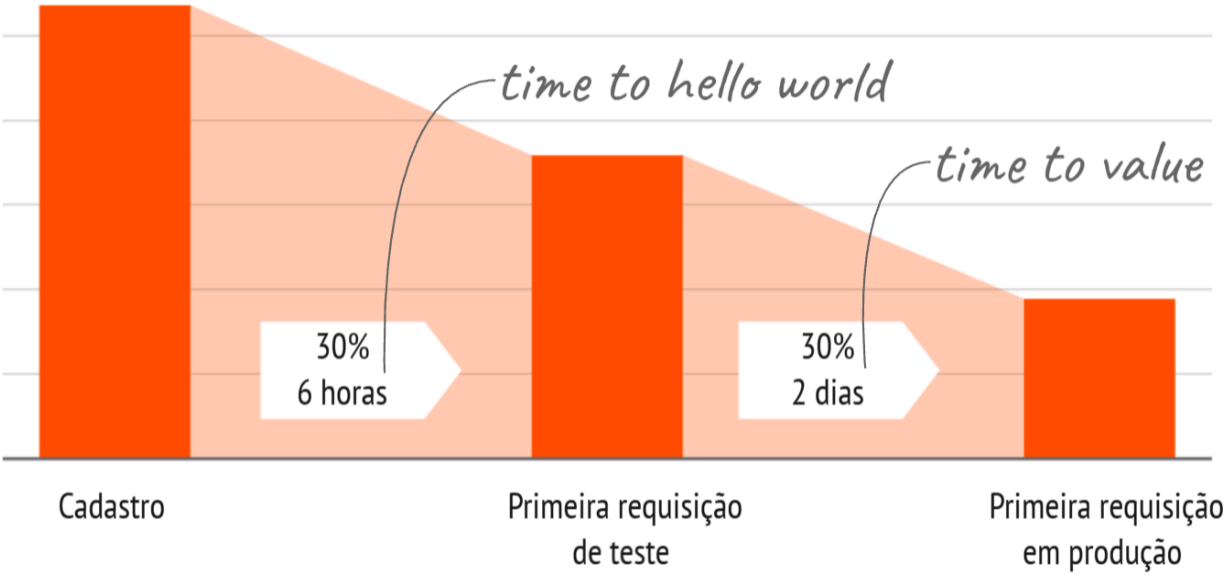


Figura 9.2: Gráfico de adoção de API. Adaptada de: Gilling, 2022.

Time to value **(TTV)**

Depois que a pessoa dev testa a API, entende como funciona e como se encaixa no app que está desenvolvendo, ela integra o app à plataforma, adapta aos casos de uso específicos e faz *deploy* no ambiente de produção.

Uma vez que o app está nas mãos do usuário final e ele o utiliza, ocorrem requisições na API (não necessariamente apenas uma) para que algo seja executado. É quando o real valor é gerado para o time do app, o *wow moment*. O TTV é o tempo entre o TTHW e o *wow moment*.

O *time to market* é a somatória do TTHW e o TTV. É preciso atenção à contabilização, porque ela pode induzir ao erro. Vamos admitir que o time do app chegou ao TTHW em 2 dias e que, antes de ir para a próxima etapa, surgiu uma demanda que levou 10 dias. Depois disso, o time retomou a integração com a API e levou 5 dias para colocar em produção. Se você não souber que a jornada foi interrompida por 10 dias, você vai acreditar que levou quase 17 dias para o *time-to market* e, considerando os fins de semana, vai

parecer que levou quase 1 mês, quando, na verdade, levou 7 dias. Um pouco mais de uma semana para quase 1 mês faz uma baita diferença na percepção de valor, principalmente se entrar na conta o *cost of delay* (custo de atraso) — quanto a empresa deixa de ganhar financeiramente enquanto o produto ou *feature* não é lançada.

> Para saber mais sobre *cost of delay*, leia o artigo *Quanto a empresa perde financeiramente quando o projeto atrasa?*, de Lucas Colucci (2017), disponível em: https://blog.plataformatec.com.br/2017/06/quanto-a-empresa-perde-financeiramente-quando-o-projeto-atrasa/.

Além da *time to hello world* e da *time to value*, outras métricas trazem mais informações sobre atritos no *onboarding*.

API calls per business transaction

Uma métrica comum de se acompanhar — e, por isso, é provável que você já a conheça ou tenha ouvido falar nela — é a *RPM* (*requests per minute*; em português, requisições por minuto). Porém, na fase de *onboarding*, é mais proveitoso medir a métrica irmã da RPM, a *API calls per business transaction* (requisições de API por transação) — a RPM será abordada em detalhes mais adiante.

Essa métrica reflete diretamente o design da API. Se um app tiver que fazer três ou quatro requisições de API para obter alguns dados dos clientes, por exemplo, significa que a API não foi elaborada prevendo certos cenários. O ideal é que o dev consiga obter o resultado desejado com a menor quantidade de requisições possível, pois essas requisições repetidas, além de afetarem a experiência do dev na integração, aumentam a propensão a erros e podem prejudicar a performance da plataforma.

Por isso, ao projetar uma API, é importante pensar em termos de transação, não no sentido financeiro, mas, sim, sobre o que o dev

usuário está tentando fazer, por exemplo, atualizar dados de contato do cliente, criar ou deletar cadastros.

As requisições que devs fazem na API também dão uma boa noção do que foi absorvido da documentação e dos exemplos de código. Se estão fazendo algo que não deveriam, talvez não encontraram a URL ou os argumentos e os métodos adequados para chamar a API. Isso mostra que é necessário modificar a documentação e/ou deixar os códigos de exemplo mais claros.

Errors per minute

Cada requisição via API recebe um código de três dígitos como resposta. As respostas que iniciam com 2 (*2xx*) indicam sucesso. A métrica *errors per minute* (erros por minuto) corresponde ao número de requisições de API com status diferente de *2xx* no intervalo de um minuto.

É importante conhecer o tipo de erro para saber o que fazer para corrigi-los. Erros *5xx* são gerados pelo sistema servidor, podendo ser problemas de conectividade, limitações de recursos ou outros problemas técnicos relacionados à garantia de qualidade e à confiabilidade da API. Os erros *4xx* são gerados pelo sistema cliente e podem ser sinal de que o dev usuário está cometendo erros devido a problemas de usabilidade da API ou falta de clareza na documentação — no capítulo *Como PMs conseguem testar APIs* passaremos pelos principais códigos de erros.

Bom, nem é preciso dizer que é essencial monitorar essa métrica de perto, não só na fase de adoção, mas também em todo o ciclo de vida da plataforma — especialmente depois de mudanças —, para que seja possível entrar em ação em tempo hábil para evitar grandes prejuízos.

As métricas relacionadas à ativação servem para dar visibilidade aos momentos, aos motivos e à frequência com que ocorrem atritos na fase de integração com a API. Uma vez que o time do app faz o

rollout (geralmente gradual) para a base de usuários finais, o tráfego na plataforma aumenta. É quando o time cliente passa a notar a performance da plataforma. Se a performance impactar negativamente a experiência do usuário final, e o time do app tiver a liberdade de migrar para uma plataforma de mercado, isso seria motivo para *churn*. Vamos, então, para as métricas que evidenciam risco de retenção.

9.3 Retenção

Em produtos SaaS, *retenção* diz respeito à regularidade com que os usuários usam o produto. Quando o usuário passa a usar o produto menos vezes ao longo do tempo, pode ser sinal de *churn* no curto prazo. Mesmo que os times que usam a plataforma não tenham a liberdade de migrar para uma de mercado, é preciso atenção. Plataforma que apresenta baixa performance, certamente, prejudica a experiência do usuário final, e este sim pode "churnar".

As métricas de retenção estão diretamente relacionadas à **saúde da plataforma** (ou *integridade*) e devemos monitorá-las para identificar problemas e orientar melhorias nas features e na performance da API, podendo ser usadas como KPIs para medir o sucesso dessas melhorias ao longo do tempo.

Request per minute **(RPM)**

Assim como as métricas *daily active users* (DAU) e *monthly active users* (MAU) — respectivamente, usuários ativos diários e usuários ativos mensais — ajudam a entender o uso e engajamento dos usuários de produtos *customer-facing*, a métrica *RPM* exibe quais apps estão realmente utilizando as APIs, ou melhor, os apps ativos (*active apps*) — métrica mencionada na seção *Ativação*.

Alto RPM pode parecer uma coisa boa, dá a impressão de que a API é bastante usada. Porém, como vimos anteriormente, uma API com design ruim pode exigir mais requisições do que o necessário. Como o objetivo do time da plataforma deve ser a eficiência, o ideal é reduzir a RPM, identificando padrões comuns e agrupando várias requisições em uma única, se possível.

Supondo que a API tem o design ideal, a RPM traz entendimento de como os times estão usando a plataforma e se ela está sendo bem-sucedida em manter usuários. Isto é, RPM que está aumentando demonstra que o uso da plataforma está crescendo. Já RPM que está diminuindo pode ser sinal de que o app conectado à plataforma está perdendo mercado ou o time cliente está migrando para outro fornecedor. Em ambos os cenários, indica possível *churn*.

Como a plataforma é interna, se a API não está atendendo bem, é provável que os times clientes se manifestem e, se resolverem usar uma plataforma de mercado, por exemplo, é provável que ocorram várias conversas. Assim, se os produtos derivados não estão perdendo mercado, nem estão migrando para outra plataforma, a diminuição de requisições pode ser indício de problemas de performance na plataforma ou problema na experiência do usuário final e eles é quem estão diminuindo o uso.

Monitorar a RPM também serve para entender como o servidor se comporta quando recebe diferentes cargas, como em momentos de pico. É importante acompanhar o volume de dados que a plataforma manipula a cada requisição. Se a plataforma recebe mais requisições do que consegue lidar ou a quantidade de dados que ela é forçada a lidar está prejudicando a performance, o time da plataforma pode escalar a infraestrutura para evitar lentidão ou inatividade.

Average and max latency

O tempo entre uma ação do usuário no app e a resposta que ele recebe a essa ação chama-se *latência* (*latency*). As métricas *average latency* e *max latency* são, respectivamente, as medidas do tempo médio e do tempo máximo que uma API leva para processar as requisições e devolver a resposta.

A latência tem impacto no tempo de carregamento do produto derivado. Altos níveis de latência podem afetar a escalabilidade e a capacidade da API de lidar com um grande número de requisições simultâneas, deixando os apps lentos. Por isso, é importante definir limites de níveis aceitáveis de latência média e máxima, para que, quando esse nível for atingido, o time da plataforma receba um alerta e entre em ação para resolver os problemas de performance antes que eles afetem a experiência dos clientes finais.

Uptime

Também conhecida como *availability* (disponibilidade), *uptime* (tempo de atividade) é a métrica padrão para medir o tempo que o serviço fica disponível para uso. Normalmente, é uma métrica de garantia de que os clientes conseguirão acessar o serviço sempre que precisarem.

É comum os serviços mostrarem o quanto garantem de disponibilidade como parte do SLA (*service level agreement* — acordo de nível de serviço), tal qual este trecho do artigo *Amazon Cognito anuncia Acordo de nível de serviço de 99,9%* (AWS, 2019):

"*Dentro de limites comerciais razoáveis, a AWS fará todos os esforços para disponibilizar o Amazon Cognito com uma porcentagem de tempo de disponibilidade mensal de pelo menos 99,9% (o "Compromisso de serviço"). Caso o Cognito não cumpra o Compromisso de serviço, você poderá receber um Crédito de serviço (...)*."

No mundo perfeito, os serviços ficariam disponíveis 100% do tempo, mas essa promessa beira o impossível. Já prometer *quase* 100% é

bastante realista. É tão normal serviços afirmarem que possuem mais de 99% de disponibilidade que isso é até uma medida, conhecida como "*nines*" (noves) — por exemplo, 99,9% de disponibilidade ou "três noves".

Os "*nines*" referem-se à porcentagem de tempo que um serviço fica disponível no período de um ano. Um sistema com 99,9% de disponibilidade tem interrupção de, no máximo, 8,76 horas por ano. A tabela a seguir mostra a quantidade de tempo que um serviço pode ficar indisponível (*downtime*) com base na quantidade de "noves".

Disponibilidade %	***Downtime* por ano**
99% (dois noves)	3,65 dias
99,9% (três noves)	8,77 horas
99,99% (quatro noves)	52,60 minutos
99,999% (cinco noves)	5,26 minutos

Alta taxa de *uptime* indica que a plataforma lida bem com o volume do tráfego. Já uma taxa baixa costuma afetar negativamente a experiência do usuário final e pode causar perda de receita. Mais disponibilidade se consegue com mais redundância de infraestrutura, mas isso significa mais custos.

CPU usage

CPU usage (uso de CPU) é a métrica que mostra o poder de processamento computacional. Alto uso de CPU indica que o servidor está sobrecarregado, processando muitas tarefas ao mesmo tempo ou que alguma tarefa está exigindo muitos recursos. Em decorrência disso, a plataforma pode apresentar lentidão, instabilidade ou até mesmo ter o serviço interrompido.

Monitorar a *CPU usage* ajuda a identificar gargalos de performance que podem ser otimizados para melhorar a eficiência da plataforma. Por exemplo, se alguma parte da plataforma está usando alta porcentagem de CPU, convém investigar se o código está causando processamento excessivo ou se há alguma coisa que possa ser feita que utilize menos recursos.

Memory usage

A métrica *memory usage* (uso de memória) é a medida da quantidade de memória que o sistema consome enquanto está em execução. O uso excessivo de memória também leva a problemas de performance e pode ser indício de problemas na arquitetura da API, como vazamento de memória ou alocação de memória ineficiente, que podem ser corrigidos por meio de *caching* ou *pooling* de conexões para reduzir a alocação de recursos. Além disso, pode ser necessário escalar a infraestrutura para lidar com o aumento do tráfego e *workload* (carga de trabalho).

É importante monitorar as métricas de integridade (saúde da plataforma) para identificar problemas o quanto antes e, se possível, atuar antes mesmo que os usuários finais sejam afetados. Afinal, quando as coisas fluem bem, o retorno costuma vir em forma de receita.

9.4 Receita

Plataformas internas são *enablers* de fluxos de receita. A receita vem dos apps que usam as APIs, portanto, a **plataforma gera receita de forma indireta**. Mesmo que indiretamente, é válido dar visibilidade do valor monetário que a plataforma capacita para comprovar sua relevância na geração de receita. É uma forma de traduzir métricas de API em KPIs de negócios.

Como exemplo, vamos assumir que em determinado mês a Order Platform processou 100% dos pedidos. Nesse período, a unidade de negócio A vendeu X e a unidade B vendeu Y. Podemos dizer, então, que graças à alta disponibilidade, performance e escala nos momentos de pico, a plataforma capacitou a geração de X+Y, ou 100% da rentabilidade.

Quando a plataforma não atua no fluxo de geração de receita, há uma maneira de atribuir valor monetário ao uso das APIs. Você saberá como no capítulo *Estrutura de custos e fluxos de receita*.

9.5 Recomendação

Tecnicamente, a fase de **recomendação não se aplica a plataformas para uso interno**. Mas existem cenários em que determinado time quer usar uma plataforma, porém é mais conveniente usar outra como intermediadora, como era comum acontecer com a Order Plataform.

Como ela é uma plataforma central, se algum time quiser usar a plataforma de pagamentos do iFood, o time da Payments Platform recomenda usá-la por meio da Order Platform. Uma vez plugado na Order Platform, o app fica dentro dos padrões de todo o iFood, facilitando as operações e a análise dos dados. Veja que a Payment Platform deixou de ganhar um cliente direto, no entanto, as transações continuam sendo feitas via Payment Platform de qualquer maneira, ou seja, a Order Platform aumenta o uso da Payment Platform.

Outro exemplo é quando algum time quer usar a Order Platform para processar pedidos. O padrão é recomendar o uso do checkout do iFood. Dessa maneira, tudo fica dentro do que foi convencionado, os riscos diminuem e o iFood tem condições de operar melhor.

O time de uma plataforma pode recomendar outra visando ganho de eficiência, mas é necessário mensurar o quanto a plataforma é recomendada? No meu ponto de vista, não, porque não é um evento que ocorre com frequência, tampouco há qualquer acionável. Porém, é importante conhecer os serviços que trabalham em conjunto para recomendar o melhor caminho quando times clientes quiserem usar a sua plataforma e, dessa forma, manter o ecossistema interno saudável.

Acredito que essa etapa do funil é dispensável, assim o funil para plataforma passa a ser como o da imagem a seguir.

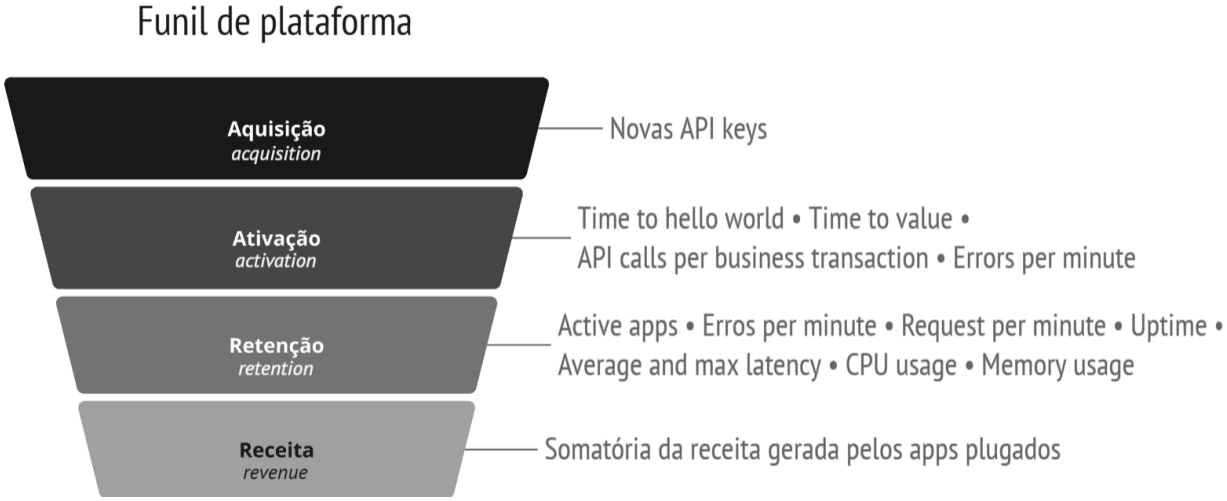


Figura 9.3: Funil de plataforma.

Observe que a métrica *errors per minute*, além de ajudar a entender a etapa de *ativação*, deve ser monitorada continuamente, por isso também está presente na *retenção*.

Quase todas as métricas que vimos podem ser usadas como objetivo a ser alcançado por times de plataforma. Por exemplo: diminuir o *time to hello world* de 15 dias para 5 dias. Já receita depende do que é vendido através dos apps, então faz sentido ser objetivo de plataformas? Vejamos adiante.

9.6 Compartilhando OKR

Todas as empresas definem metas para direcionar onde pretendem chegar em determinado período. Algumas usam o framework de OKR e pode ser difícil compartilhar objetivos de plataforma com outras áreas da empresa.

> Se OKR é um assunto novo para você, recomendo a leitura do post *OKR e KPI: definições e diferenças* no blog do Raphael Albino, disponível em https://rapha-albino.com.br/okr-e-kpi-definicoes-e-diferencas/.
>
> Agora, se a dificuldade é aplicar o framework, é mais indicado o post *Aplicando OKR de forma efetiva*, disponível em: https://rapha-albino.com.br/aplicando-okr-de-forma-efetiva/.

Em teoria, plataformização é, por si só, uma estratégia da empresa. Portanto, seria mais coerente se plataformas tivessem seus próprios OKRs no nível estratégico. Na prática, o que presenciamos é um enorme desentendimento, por dois motivos:

1. Crença de que os objetivos da plataforma precisam ser atrelados a métricas de negócio para que tenha valor;
2. Experiência restrita a medidas de TI tradicionais orientadas a projetos, que, em vez de vincular o uso da API aos KPIs de negócios, vincula a entregáveis ou *milestones*.

Se o contexto em que você vive é o primeiro caso e é possível vincular o tipo de plataforma que você atua à geração de receita, ao crescimento da base de clientes, à retenção ou aumento de consumo pelo cliente, então pode ser uma boa compartilhar o OKR com outras áreas, por exemplo, com o marketing. Mas cuidado com o KR (*key result*). Se o KR do marketing é aumentar em 50% a aquisição de clientes, o do time da plataforma não deveria ser o mesmo, porque o time não tem controle sobre as campanhas. O

time da plataforma pode entrar com o KR de **suportar** 50% mais tráfego de usuários que o marketing estima atrair.

Para fazer um paralelo, quero trazer outra plataforma do iFood, a Faster. Ela é dividida em quatro frentes, e a de Communication, uma das que atuei, viabiliza o desenvolvimento de canais de comunicação: *push notification*, e-mail, SMS, entre outros.

Um dos objetivos dessa plataforma era compartilhado com o marketing: conversão via *push*. É certo o time da plataforma compartilhar o KR de conversão? Não. O time da plataforma não cria a campanha, não cria o texto, não define o horário dos disparos. A conversão de vendas depende de variáveis fora do controle do time da plataforma, por isso o time não deve assumir que o envio de *pushes* vai aumentar o número de pedidos no app do iFood. A responsabilidade do time da plataforma é garantir que a *feature* de disparo dos *pushes* funcione adequadamente.

Então, vamos supor que uma pessoa do marketing configurou um envio de *push* para conversão de almoço, a ser iniciado às 11h e finalizado às 11h30. A plataforma enviou os pushes e finalizou às 16h. Entrega de refeição é sensível ao tempo, o pedido não leva dias, leva minutos. Então, imagine você receber o *push* às 16h com um texto falando sobre almoço. Incoerente, não é mesmo?

Se o objetivo era terminar o disparo, no máximo, às 11h30 para que houvesse tempo hábil para conversão e a plataforma *bugou* ou *engargalou*, e não conseguiu terminar o envio dos *pushes* dentro do horário determinado, a falha é da plataforma. Então, o que deve ser feito para aumentar a capacidade e distribuir esse volume de *pushes* até 11h30? Isso está sob o controle do time da plataforma, é uma métrica acionável pelo time; a conversão dos *pushes* em pedidos não é, por isso não pode ser considerada a métrica de sucesso desse time.

O inverso também é verdadeiro. Vamos assumir que determinada funcionalidade do app fica lenta nos momentos de pico e, devido a

isso, está aumentando o *churn*, cuja causa raiz é o design da arquitetura da plataforma. O time do app tem duas opções: usar outra plataforma ou desenvolver a própria solução. Ambas significam *churn* da plataforma e, se a estratégia de plataformização é a reutilização, é pouco provável que a alta gestão aceite qualquer uma dessas opções. Então, só resta mudar a arquitetura da plataforma, e isso está fora do controle do time do app.

Neste caso, o objetivo da plataforma até poderia ser "diminuir o *churn* do app", porém, pode haver outros motivos pelos quais os usuários estão abandonando o app. O mais prudente seria "aumentar a disponibilidade de 99,99% para 99,999%" — aumentar de quatro noves para cinco noves. O time da plataforma analisaria como chegar lá e, se o time do app não tiver nada a fazer, as duas squads nem precisam compartilhar esse objetivo.

Outro cenário abrange os casos em que não é possível vincular objetivos da plataforma aos de negócio. Neste contexto, como PMs, nós é que precisamos mudar de mentalidade e usar a "régua" apropriada ao contexto que se apresenta. Plataformas proporcionam capacidades, por isso, é normal que os objetivos sejam entregas de *features*. Por experiência própria, é melhor trabalhar visando a entregas de *milestones* do que receber responsabilidade sobre algo que o time da plataforma não tem controle. Por isso, apesar de tudo que você já ouviu sobre gestão de produto orientada a *outcome*, *output* não é necessariamente nocivo na gestão de plataformas.

Agora, se é possível compartilhar OKR com outras áreas e isso não é feito devido ao mero hábito de seguir como projeto de TI, você terá que evangelizar. Até lá, busque ao menos não comprometer o objetivo da plataforma com atividades fora do controle do time.

9.7 Juntando tudo

As métricas ajudam a provar valor, mas, dependendo do tipo de plataforma, isso não é possível. Para piorar, pode ser que o time da plataforma nunca receba a devida atribuição pelo sucesso que os times usuários obtiveram do uso da plataforma. Por exemplo, os times dos apps podem executar testes A/B nos quais as variantes são construídas em cima da mesma plataforma e, como ela está oculta, raramente o sucesso será atribuído à plataforma que possibilitou esses experimentos.

Independente de a quem ou ao que o sucesso é atribuído, o insucesso por falta de recursos certamente recai sobre a plataforma. As métricas fornecem informações valiosas sobre a saúde dela, por isso devemos monitorá-las para garantir disponibilidade, confiabilidade e performance, identificar problemas e tomar decisões com base em dados sobre o que melhorar para atender às necessidades dos times usuários e proporcionar uma boa experiência aos usuários finais.

Para isso, é essencial que haja um dashboard que o time da plataforma possa visualizar a qualquer momento. Considerando quais métricas seriam importantes estar nesse dashboard, preenchemos o *Lean Canvas*.

Figura 9.4: Lean Canvas – Métricas.

Repare que, assim como nos outros quadrantes, as métricas foram associadas a cada perfil de público — definido no segmento de clientes — para dar visibilidade de qual deles sofre o impacto da percepção de valor.

Ainda para sustentar a proposta de valor, há a *Estrutura de Custos* e os *Fluxos de Receita*, os dois últimos quadrantes do *Lean Canvas*, que vamos explorar no próximo capítulo.

Capítulo 10
Estrutura de custos e fluxos de receita

Uma empresa pode até oferecer uma solução mediana e não ter uma proposta de valor clara, mas é impossível fechar os olhos para os aspectos financeiros. Por mais que não se faça as contas na ponta do lápis, é improvável que não se perceba que o dinheiro está diminuindo ao longo do tempo.

Os quadrantes *Estrutura de Custos* e *Fluxos de Receita* do *Lean Canvas* trazem visibilidade de como será a precificação e o modo de cobrança para equilibrar a balança do que sai de dinheiro.

Um dos maiores valores que um produto pode ter é a sua receita. Mas como mostrar isso no caso das plataformas, se elas não geram receita diretamente? No capítulo sobre métricas, vimos que uma possibilidade é contabilizar o total da receita indireta que a plataforma dá a capacidade de gerar. E quando isso não é persuasivo suficiente?

Uma saída é mostrar o quanto a plataforma está ajudando a empresa a economizar; outra é dar visibilidade dos custos de cada app plugado na plataforma, para mostrar que o custo que parece ser da plataforma é, na verdade, das unidades de negócios. E por que mostrar que o custo não é da plataforma sozinha? Porque manter a plataforma pode ser caro demais.

10.1 Infraestrutura é caro

Quando a empresa é pequena, possui um ou dois apps no mercado que trafega pouco volume de dados, exige pouco de infraestrutura. Mas quando ela cresce, esse cenário muda brutalmente. O *print* a seguir mostra os preços em dólar do *Amazon API Gateway* fornecido

pela AWS. Parece um preço simbólico, mas para empresas que trafegam megadados — "*dados multivariados e de elevada dimensão, geralmente criados em tempo real, e apresentam um crescimento exponencial (na escala temporal)*" (WIKIPÉDIA, c2024), APIs podem receber milhares de requisições em questão de minutos cuja somatória vai para milhões de dólares.

Figura 10.1: Preço do Amazon API Gateway. Fonte: AWS, c2023.

David Heinemeier Hansson (2023), cofundador da 37signals, empresa responsável pelo BaseCamp e Hey, postou no Twitter que os serviços da AWS — S3, RDS, OpenSearch e Elasticache — somaram mais de US$ 3,2 milhões em custos para a empresa em 2022. Isso porque nem tudo o que produzem ou todas as tecnologias que usam estão na AWS. Imagine se estivessem.

Seria o alto custo de grandes operações o motivo pelo qual empresas como Uber apresentaram prejuízo, apesar do número de viagens aumentar continuamente? De acordo com um artigo

publicado no Tecnoblog (ALECRIM, 2022), o Uber teve prejuízo de US$ 1,2 bilhão no primeiro trimestre de 2022.

A história fica mais dramática para empresas que operam no Brasil. No *print* anterior, foi selecionada a região Norte da Virgínia. Repare, no próximo *print*, o aumento dos preços para São Paulo, região ideal para atender aos brasileiros — porque quanto mais próximo o servidor estiver do usuário, menor a latência. Então, além de pagar quase 6 vezes mais devido ao valor do dólar no Brasil, as empresas pagam mais caro pelos serviços para proporcionar melhor experiências aos usuários.

Figura 10.2: Preço do Amazon API Gateway em São Paulo. Fonte: AWS, c2023.

O diretor de pesquisa da Gartner, Henrique Cecci, em entrevista publicada no Portal IPNews (2022), afirma que o custo de serviços *cloud* é 78% maior no Brasil do que nos Estados Unidos. Ele diz que a Gartner estima que mais de 50% do que se gasta com *cloud* é

devido à ineficiência e que, no Brasil, esse número pode ser ainda pior.

Claro que, quanto mais robusta for a plataforma, maior será o custo. Mesmo que a base de usuários se mantenha a mesma, o uso do produto gera dados, que são somados ao que já está armazenado. Isso significa que acumula cada vez mais, o que leva ao aumento de custo por usuário ao longo do tempo, sem que a receita aumente. Por isso, são necessárias políticas de governança, como limite de tempo para manter os dados armazenados ou modelo de precificação com base em volume de dados, por exemplo. Mas, como dito no começo do capítulo, os custos não são apenas da plataforma, e precisamos deixar isso claro.

10.2 Equilibrando custo com a receita

Vamos imaginar uma plataforma que, assim como a 37signals, desembolsa US$ 3,2 milhões por ano com serviços da AWS — arredondando, US$ 270 mil por mês. Quando a alta gestão olha para esses milhões de dólares, ela tem a interpretação de que a plataforma custa caro, o que confronta na percepção de valor. Mas esse custo não corresponde à realidade. Na verdade, o custo de diversos apps estão centralizados na conta da plataforma.

Para que o valor da plataforma não seja questionado, uma das alternativas seria separar quanto a plataforma custa para se manter e o custo de cada app conectado. Essa conta é difícil e pode ser imprecisa.

Por exemplo, poderíamos usar as requisições na API — a métrica RPM — como base para cálculo. Então, vamos supor que há três apps conectados a tal plataforma que custa US$ 270 mil por mês e, na média, 50% das requisições vêm do app A, 30% do app B e 20% do app C. Dessa forma, poderíamos calcular proporcionalmente

quanto esses apps custam quando a requisição bate no Amazon API Gateway. Sabendo quais dessas requisições salvam dados no banco, também seria possível calcular o custo de cada um com S3, por exemplo.

Mas não é bem assim. Se a plataforma salva outros dados gerados por ela junto com os dados enviados pelo app, separar o montante de cada um é trabalhoso e, se a estrutura da plataforma for grande, pode trazer mais ruído do que informação. Uma saída mais simples é precificar o consumo.

Como precificar o consumo

Há duas opções para precificar o consumo. Uma delas é simplesmente dividir o valor total, proporcionalmente, pela quantidade de requisições de cada app.

Voltando ao exemplo da plataforma que custa US$ 270 mil por mês, a transferência do custo para cada app seria como mostra a tabela.

App consumidor	Requisições	Custo
A	50%	US$ 135 mil
B	30%	US$ 81 mil
C	20%	US$ 54 mil
Total	100%	US$ 270 mil

Seria justo? Sim, afinal, as empresas colocam os custos e despesas no preço. Por que não poderia se fazer o mesmo para a plataforma? Se ela não existisse, os times teriam custos com infraestrutura de qualquer maneira, com a desvantagem de o custo ser maior se comparado ao preço de "atacado", como mostram os *prints* da AWS.

Esses números são suficientes para constatar se o saldo do faturamento de cada app é positivo ou negativo subtraindo o custo, o que serve de insumo para a alta gestão decidir se vale a pena mantê-los.

Outra forma de precificar, e até mais interessante, é com base no preço das plataformas de mercado que oferecem o mesmo tipo de serviço. Vamos supor que se trata de uma plataforma que registra os eventos acionados quando o usuário final interage com o app, como o Segment (https://segment.com/).

O *print* a seguir mostra que o plano Team custa US$ 120 por mês para até 10 mil visitantes únicos (logados ou anônimos) — cada visitante custa US$ 0,012.

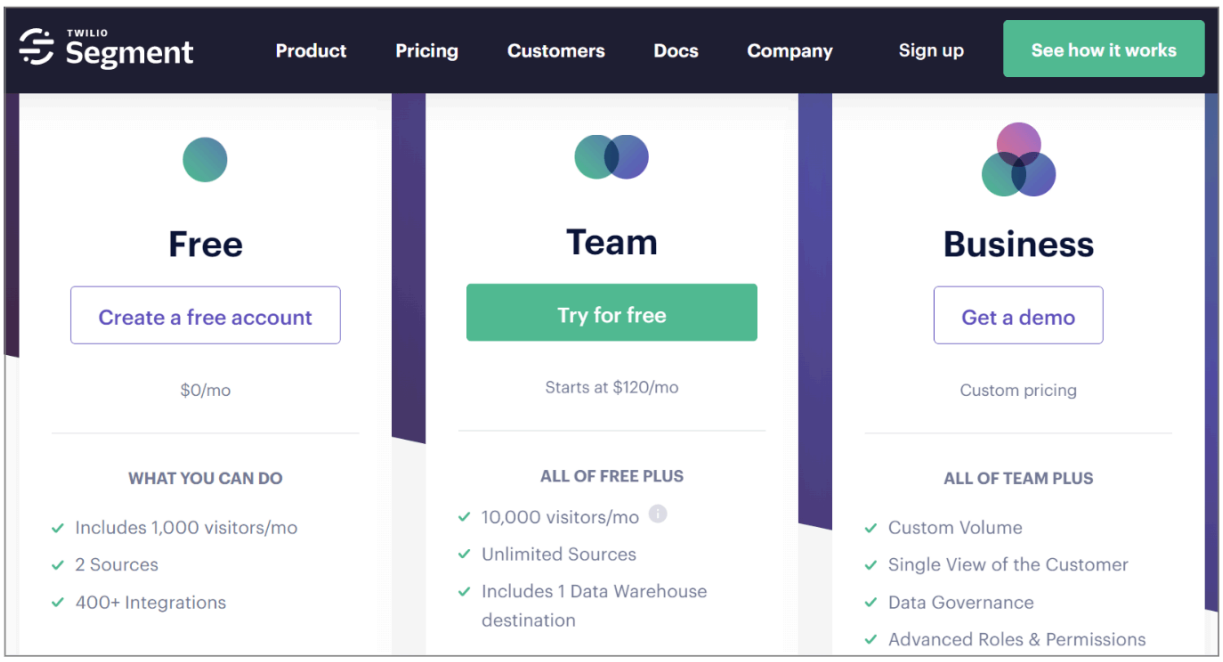


Figura 10.3: Preço do Segment. Fonte: Segment, c2023.

A ideia é colocar em um dashboard o valor que a empresa gastaria usando o Segment e quanto ela está gastando usando a plataforma interna, fazendo aquela conta básica de dividir proporcionalmente ao consumo, como vimos antes.

Então, ainda seguindo o exemplo, vamos admitir que a quantidade de visitantes únicos é proporcional à porcentagem de requisições para facilitar o cálculo. Assim, seria respectivamente: 50 milhões, 30 milhões e 20 milhões de visitantes únicos, cujos preços seriam como mostra a tabela a seguir.

App consumidor	Visitantes únicos	Plataforma interna	Segment	Econon
A	50 milhões	US$ 135 mil	US$ 600 mil	US$ 465 mil
B	30 milhões	US$ 81 mil	US$ 360 mil	US$ 279 mil

App consumidor	Visitantes únicos	Plataforma interna	Segment	Econon
C	20 milhões	US$ 54 mil	US$ 240 mil	US$ 186 mil
Total	100 milhões	US$ 270 mil	US$ 1,2 bilhões	US$ 930 mil

A coluna *Plataforma interna* refere-se ao cálculo exibido na tabela anterior, mas aqui é relacionado ao número de visitantes de cada app, em vez de requisições. A coluna *Economia* é o quanto a plataforma ajudou a economizar, comparado ao uso do Segment. A diferença de US$ 930 mil é brutal — em reais, esse valor é colossal. Essa forma mostra claramente o quanto a plataforma está ajudando a empresa a economizar. Não é uma forma interessante de elevar a percepção de valor da plataforma?

É óbvio que, na prática, a diferença de valor não será necessariamente gritante como a do exemplo. Pode até ser que a plataforma saia mais caro — difícil, mas pode acontecer. Se o preço fictício da plataforma for mais alto do que as de mercado, o time da plataforma pode avaliar o que fazer no design da arquitetura e das APIs para baixar o custo e, se nada for suficiente, a alta gestão pode repensar se vale a pena manter a plataforma restrita ao uso interno ou se o caminho é abrir as APIs para o público externo e monetizar.

10.3 Juntando tudo

Plataformas devem ajudar a empresa a ser eficiente, e custos não são apenas uma questão de resultado, são também uma medida de eficiência. Os times dos apps se ocupam de validar hipóteses o mais rápido possível, por isso, dificilmente dedicarão tempo para otimizar código e arquitetura visando controle de gastos.

Esse é o papel do time da plataforma, e centralizar em um local garante que um único time se concentre em evitar gastos desnecessários com infraestrutura e qualidade de software — o que é bem diferente de diversos times terem que se preocupar com isso.

Isso traz de volta a importância do design da arquitetura e a escolha dos serviços complementares — tipo de servidor e de banco de dados, por exemplo —, que vimos no capítulo *Proposta de valor*, pois são coisas que podem otimizar o investimento e trazer maior margem de lucro.

Então, para preencher o quadrante *Estrutura de Custos* do *Lean Canvas*, vamos usar o caso da 37signals que escolheu para a arquitetura os seguintes serviços da AWS: S3, RDS, OpenSearch e Elasticache. Não há necessidade de colocar os valores, porque eles variam de acordo com o uso.

Figura 10.4: Lean Canvas – Estrutura de Custos.

Já para o quadrante *Fluxos de Receita*, você pode colocar o que faz mais sentido para o tipo de plataforma: a simples transferência dos custos para cada app; preço com base em plataformas similares do mercado; ou, como visto no capítulo *Métricas*, a receita indireta gerada através da plataforma.

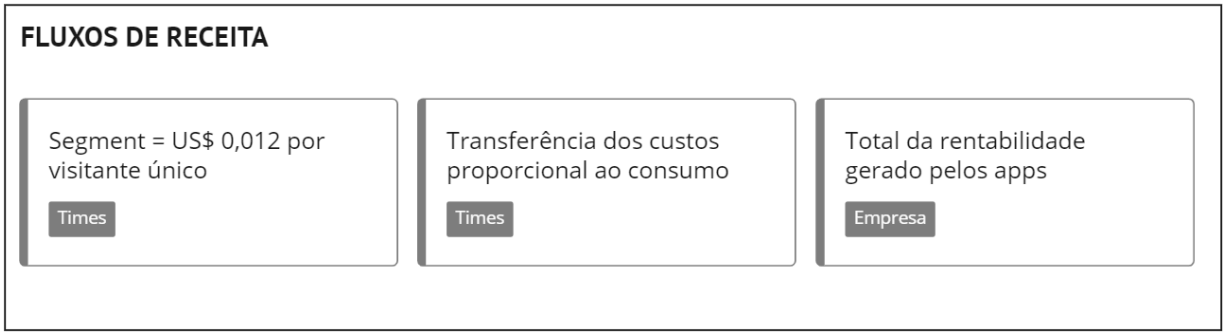


Figura 10.5: Lean Canvas – Fluxo de Receita.

Lembre-se de que o *Lean Canvas* é uma ferramenta para visualizar o que já foi e o que ainda precisa ser validado. Então, você pode colocar as hipóteses de receita fictícia e depois entender qual faz mais sentido. Quando uma delas se tornar certeza, você pode adicionar alguma tag que mostre a opção "validada" ou simplesmente deletar as refutadas — o mesmo serve para os demais quadrantes.

E assim, com o *Lean Canvas* para plataforma completo, chegamos ao final da Parte 2 do livro. A essa altura, é provável que você já tenha algumas certezas, mas ainda tenha coisas para validar. Então, partiu *discovery*?

PARTE 3 — Discovery de plataforma e API

Capítulo 11
Em busca de eficácia

O preenchimento do *Lean Canvas* torna visíveis hipóteses que estavam implícitas e, antes que qualquer coisa seja construída, é prudente validar essas hipóteses com o intuito de obter "*confiança de que o que se acredita que acontecerá é o que realmente vai acontecer*" (PATTON, 2016; tradução livre), o conhecido *discovery*.

Acontece que o *discovery* está perdendo credibilidade, e para plataformas é ainda pior. Percebo pelo menos dois motivos por que isso acontece: **herança do modelo cascata (*waterfall*)** e **má interpretação do processo**.

11.1 Herança do modelo cascata

No modelo cascata, o processo de desenvolvimento de produto de software começa a partir da ideia de algum executivo, cliente ou stakeholder. Dessa ideia, desenvolve-se o produto completo para, então, vendê-lo. O esforço de vendas é direcionado ao marketing e ao comercial. Basicamente, o processo é:

1. A ideia surge;
2. Uma pessoa entende os requisitos, descreve uma documentação (extensa) — que pode levar meses para ficar pronta — e passa para uma pessoa fazer o design da interface do software;

3. A pessoa designer define o layout com base nos requisitos contidos na megadocumentação — o que também pode levar meses de trabalho — e passa para as pessoas de desenvolvimento;
4. As pessoas devs desenvolvem o software, que pode levar anos para ficar pronto (isso mesmo, anos!), também com base nos requisitos que foram definidos há meses;
5. Depois de anos, o software está nas mãos dos clientes.

Neste modelo, o software é visto como um *projeto* com começo, meio e fim, mas o final nem sempre é feliz. Há infinitos casos em que os clientes se queixam de que demorou muito mais do que o prazo combinado; que, depois de tanta "coisa", não é o que esperavam; que o software não faz o que precisam, que não funciona direito, não conseguem usar, e por aí vai.

Desenvolvimento de software é projeto

Por décadas, o processo de criação e desenvolvimento de software foi baseado no modelo de gestão de projetos da construção civil, cuja premissa é a de que um bom planejamento garante que a obra cumprirá determinado orçamento, será concluída com determinada qualidade e em um determinado prazo. Para que isso seja verdade, basta seguir o plano e executar as etapas prescritas com *eficiência*.

No desenvolvimento de software, isso se traduziu em: "*qualidade é cumprir os requisitos e sucesso é a entrega do escopo acordado, dentro do prazo e do orçamento*" (VALE, c2021). Nesse caso, assume-se que a solução predefinida no planejamento do projeto é o que certamente resolverá o problema. Isso significa que o foco é a *solução* — não o *problema* — e o escopo é fechado, isto é, sem considerar que pode haver mudanças. Assim, o objetivo torna-se entregar coisas, muitas coisas, o máximo de coisas possível no menor tempo. Basta *eficiência* na entrega do que foi acordado entre as partes.

No entanto, muitas vezes, o que se tem no final são partes de software que formam um todo desconexo, inconsistente, que não funciona bem; tão ruim quanto é cumprir à risca todos os requisitos, o prazo, o orçamento, e descobrir, no final de tudo, que o software até funciona bem, mas não resolve o problema que o cliente disse ter no começo do projeto. Como o cliente não obtém o resultado esperado ao usar o software, é preciso fazer mais coisas para suprir o que o software faz e o que deveria fazer. Conclusão: tem muita coisa feita, mas nada funciona plenamente, o que aumenta o backlog.

Esse efeito é inevitável no **gerenciamento de projetos em cascata**, porque as pessoas não se dão conta de que tudo o que veem como certeza, na verdade, não passa de suposição e, por isso, repleta de incertezas.

O começo de um projeto é o momento em que "*há a menor quantidade e a menor qualidade das informações e do conhecimento sobre o que de fato precisa ser feito*" (VALE, 2020) e reflete em tudo o que virá. Então, ao longo do desenvolvimento, as verdades vêm à tona e, como consequência, a constante mudança do escopo inicial, pois percebe-se que o que foi planejado não é exatamente a coisa certa a ser feita, ou até mesmo é inviável. Eis que surge o Manifesto Ágil.

Manifesto Ágil e eficiência

O Manifesto para Desenvolvimento Ágil de Software (BECK *et al.*, 2001) trouxe uma nova forma de desenvolver software, em que atender às necessidades dos clientes passa a ser o centro — não a solução de escopo fechado —, e o processo favorece adaptação às mudanças que ocorrem durante o desenvolvimento, por meio de entregas frequentes de software funcionando como forma de obter feedback contínuo.

O manifesto põe a perspectiva de entrega de valor em evidência, colocando-a como mais importante do que se ater ao que foi

definido no início do projeto. O manifesto sugere que a entrega de valor deve ser em fases para obter feedback em ciclos curtos, não somente depois que 100% do software estiver pronto. Isso mostra que o objetivo é *desenvolver a coisa certa*.

E o que mudou na forma como novas demandas chegam para os times? Nada. O manifesto diz que "*mudanças nos requisitos são bem-vindas, mesmo tardiamente no desenvolvimento*" (BECK *et al.*, 2001), mas não menciona a necessidade de evidências que mostrem que a demanda resultará em algo que será de fato usado.

Então, o padrão (desde sempre) é a demanda chegar em forma de solução, como "preciso da funcionalidade X", o que, na mente de quem demanda, é o que resolverá determinado problema — e é comum vermos que a tal funcionalidade nem é usada.

Dessa forma, pouca coisa muda em comparação com o modelo em cascata. A ideia surge e o time faz entregas menores, incremental e/ou iterativa — o que dá a entender que está se seguindo alguma metodologia ágil. Mas qual é o custo de se construir algo que nem deveria ser construído, mesmo que em um ciclo mais curto? Qual a vantagem de ser eficiente na construção da coisa errada?

Claro que descobrir em um ciclo curto que a solução não é tão boa para os clientes como se imaginava é melhor do que descobrir isso somente no fim de um projeto inteiro. Mas pode acontecer de descobrir, a cada entrega, que a "coisa" nem deveria ter sido construída, simplesmente porque não resolve o problema apropriadamente, por exemplo. Isso mostra que **eficiência** não significa **eficácia**.

EFICIÊNCIA VERSUS EFICÁCIA

Peter Drucker (1963; tradução livre), considerado o pai da teoria da administração moderna, definiu que **eficiência é fazer as coisas do jeito certo**; já **eficácia é fazer a coisa certa**.

Veja que não ter a funcionalidade X não é o problema. O problema está por trás daquilo que parece que será resolvido pela funcionalidade X — e talvez resolva mesmo! Mas, e quando não resolve? Dizem que o cliente não sabe o que quer e trazem exemplos de que se Henry Ford tivesse perguntado aos clientes o que eles queriam, eles teriam dito "cavalos mais velozes"; ou a frase do Steve Jobs, "as pessoas não sabem o que querem até você mostrar a elas". Sabe-se lá se essas frases são de autoria deles, mas seria isso mesmo?

Dependendo do quão inovador é o produto, é capaz de as pessoas realmente não saberem que precisavam dele até tê-lo em mãos. Contudo, arrisco dizer que, na verdade, as pessoas não sabem expressar o que precisam (ou desejam), muito menos possuem repertório para saber o que é possível ser feito. Não é papel do cliente saber expressar o que ele acredita que quer. É responsabilidade das pessoas PMs descobrirem o produto certo a ser construído (*discovery*) e entregá-lo ao mercado (*delivery*). É como Alisson Vale, referência na comunidade ágil, escreveu no livro *A Fórmula da Eficácia*:

"(...) *'o cliente pede, nós fazemos'. Isso significa que a dimensão de eficácia (fazer a coisa certa) é tratada como uma responsabilidade dele. É um modelo onde o sucesso do projeto depende do cliente definir bem o que deve ser feito e de nós (equipe) executarmos bem o que ele nos pede. Como se o ganho da ação dependesse do advogado executar bem a estratégia de defesa formulada pelo seu cliente; ou como se a cura dependesse do médico receitar o remédio que o paciente quer tomar*" (VALE, 2020).

Problemas podem ser resolvidos de várias maneiras. Pode haver solução melhor do que a que o cliente conhece e, por meio do processo de *discovery*, entenderemos qual é o problema para então chegarmos à solução desejada e que funciona — a **eficaz**, aquela que proporciona o *problem-solution fit*.

Assim, um pouco depois do Manifesto Ágil, surgiu uma nova forma de se desenvolver software, propagada pelo livro *A Startup Enxuta*.

Startup enxuta e eficácia

Startups seguiam os processos já conhecidos — ágil ou em cascata —, com a diferença de que nem tinham clientes e, sem dinheiro para marketing, tinham dificuldades para atrair usuários. Elas nasciam, ninguém aparecia para usar o produto e acabavam por ficar abandonados no oceano infinito que é a internet.

Eric Ries (2012) diz que startups fracassam devido à "*fascinação associada a um bom plano, a uma estratégia sólida e a uma pesquisa de mercado completa (...), mas isso não funciona, pois as startups operam com muita incerteza. As startups ainda não sabem quem são seus clientes ou como devem ser seus produtos. À medida que o mundo fica mais incerto, é cada vez mais difícil prever o futuro*". Então, ele propõe associar o *Lean* (enxuto) do sistema criado pela Toyota para produção de carros — e que a comunidade ágil já havia adotado — ao que ele nomeou de *aprendizagem validada*, que nada mais é do que o método científico: levantar hipóteses e, por meio de experimentos, testá-las de modo a constatar empiricamente se elas são verdadeiras ou falsas.

No ponto de vista dele, o aprendizado é o cerne do desenvolvimento de um negócio sustentável que, inicialmente, atua sob condições de extrema incerteza. Ries (2012) afirma que "*devemos aprender a verdade acerca de que elementos da nossa estratégia estão funcionando para concretizar nossa visão, e quais são apenas absurdos. Devemos aprender o que os clientes de fato querem, e não o que eles dizem que querem ou o que achamos que devam querer*".

Assim como nas metodologias ágeis, o tempo total do ciclo de feedback entre *construir-avaliar-aprender* deve ser minimizado. É aí

que entra o MVP, que consiste em desenvolver o mais rápido possível um produto mínimo viável, com o mínimo de esforço, no menor tempo, e colocá-lo diante dos possíveis clientes para avaliar a reação deles perante a eficácia do produto, e até mesmo tentar vender a eles. Com base no feedback dos clientes, vem a decisão: "*pivotar a estratégia original ou perseverar*" (RIES, 2012).

PMs já estão bem familiarizados com essa parte, porém, no ambiente corporativo, há apego ao processo, e eu percebo que alguns conceitos são mal interpretados.

11.2 Má interpretação de alguns conceitos

Alguns termos e falas realmente abrem margem à má interpretação. Por exemplo, o simples fato de a metodologia se chamar "ágil" dá a entender que está associada à velocidade; ou a afirmação de Eric Ries: "*se o objetivo fundamental do empreendedorismo é se envolver na construção de organizações sob condições de extrema incerteza, sua função mais essencial é a aprendizagem*" (RIES, 2012). Ele também diz que o modo correto de se avaliar a produtividade é quantificar a aprendizagem que foi validada e "*não em termos de quanta coisa estamos desenvolvendo*".

É verdade que a quantidade de coisas que se desenvolve não está relacionada ao sucesso, mas dizer que o objetivo do empreendedorismo é a aprendizagem, e que ela é que deveria ser quantificada, é demais. Que tal levar dois anos entre conhecer bem os clientes e lançar o MVP no mercado? Agora imagine se, depois de dois anos, se descobre que a hipótese é falsa. Em que ajudou um *discovery* longo na execução da estratégia errada?

É preciso saber o quanto antes que a estratégia está errada — o famoso "errar rápido". Eric Ries é claro quando fala sobre ser *lean* e as validações serem baratas e rápidas. Mas, e quando se erra rápido

centenas de vezes seguidas? Por exemplo, em dois anos você validou 200 hipóteses que se mostraram falsas e, até agora, não tem produto nas mãos dos usuários. A somatória desses aprendizados torna-se longa demais. Será que Eric Ries acredita que valeu a pena ter só aprendido? Aposto que, na visão dos investidores, foram desperdiçados dois anos em tempo e dinheiro.

Toda hipótese que se mostra falsa traz resultado apenas para a própria descoberta, não gera resultado concreto para o negócio. Como PMs, nosso objetivo primordial é gerar resultado para o negócio, certo? O *discovery* é o meio de encontrar a solução que vai trazer o resultado. Imagine descobrir de primeira que a hipótese é verdadeira. É ou não é infinitamente melhor do que ficar rodando em círculos testando absolutamente tudo?

Discovery suficiente para diminuir riscos

Penso diferente do Eric Ries: o objetivo de empreender não é o aprendizado. Infelizmente, diante de incertezas, é preciso aprender algumas coisas e esse aprendizado é como um pedágio que se paga para evitar que se avance na direção errada. Se existisse uma bola de cristal que respondesse qual a coisa certa a se fazer, qualquer pessoa que empreende certamente a usaria em vez de ter que se deparar com falhas ao longo do caminho. Assim, o objetivo final é o resultado para o negócio de modo que seja sustentável, coisa que só é possível estando no caminho certo, e uma das maneiras de estar no caminho certo é diminuir incertezas e riscos.

Veja que diminuir incertezas e riscos não significa eliminar cada um deles, imaginando todos os cenários possíveis que levam à *analisis paralisis*. Eliminá-los por completo toma tempo demais e, no fim das contas, custa caro.

Analisis paralisis

"*Paralisia por análise (também paralisia de análise ou da análise) descreve um processo individual ou em grupo onde analisar demais ou pensar demais em uma situação pode fazer com que o movimento ou a tomada de decisão da situação se torne "paralisada", o que significa que nenhuma solução ou curso de ação é decidido dentro de um período de tempo natural. Uma situação pode ser considerada muito complicada e uma decisão nunca é tomada, ou tomada muito tarde, devido à ansiedade de que um problema potencialmente maior possa surgir*." (WIKIPEDIA, c2023).

Então, como selecionar as hipóteses a serem validadas? As melhores candidatas são aquelas que apresentam riscos potenciais de levar a ideia ao fracasso — aquelas críticas, em que, se o risco se concretizar, todo o resto falha junto com elas. Os riscos que Marty Cagan (2019) sintetiza em quatro categorias também servem para plataformas:

- **Risco de valor** (*value risk*): os times têm interesse em usar?
- **Risco de usabilidade** (*usability risk*): os devs saberão como usar?
- **Risco de viabilidade técnica** (*feasibility risk*): é possível desenvolver com as habilidades e tecnologia disponíveis, dentro de um determinado prazo?
- **Risco de viabilidade do negócio** (*business viability risk*): funciona para o negócio (vendas, marketing, finanças, jurídico etc.) em termos de investimento, retorno e operações?

Vale ressaltar que não necessariamente haverá riscos relevantes em todas as categorias. No entanto, é importante investir tempo também em descoberta quando não há consenso sobre a solução para evitar problemas de usabilidade e/ou viabilidade técnica. Se as pessoas do time consideram baixo o risco de valor, usabilidade,

viabilidade técnica e de negócio, não há necessidade de *discovery*, segue-se diretamente para o *delivery*, com a consciência de que há margem para erro — é preferível errar e corrigir em curto espaço de tempo a ser um time extremamente conservador que tenta exterminar todas as incertezas e riscos.

Outro fator a se considerar é o *tempo*. Se o tempo para o *discovery* for maior do que o de *delivery*, compensa ir direto para o desenvolvimento. Nada, absolutamente nada traz mais certeza do que software funcionando nas mãos dos usuários. Até lá, não importa o que ou como, não importa se está na fase de *discovery* ou *delivery*: **tudo o que se tem são hipóteses**.

Discovery bom é aquele que valida de forma rápida e barata, ou seja, um experimento menos custoso do que o desenvolvimento. Em vez de perseguir a perfeição, coloca o MVP no mercado e, ao adquirir respostas concretas, tira-se o que está errado.

Lembre-se: a finalidade do *discovery* e de endereçar os riscos é descobrir a coisa certa a se construir — a solução valiosa, usável, factível e viável, a **eficaz** sob o ponto de vista do usuário e do negócio. Isso não significa fazer muitas coisas.

Fazer muitas coisas em pouco tempo NÃO é agilidade

Como vimos no começo do capítulo, o mundo do desenvolvimento de software herdou conceitos de projetos da construção civil, e não para por aí. A coisa mais comum de se presenciar é o vício do modo de trabalho com a mentalidade histórica da era industrial.

Na indústria de manufatura, a produção é previsível. Milhares de unidades do mesmo produto devem ser exatamente iguais, a mínima diferença é defeito. O modelo industrial persegue a eficiência — melhorias no processo para fazer o máximo possível no menor tempo — sem alterar o produto final. Máquina parada é sinônimo de desperdício.

O desenvolvimento de produtos digitais é o oposto. Cria-se coisas novas de maneiras diferentes. Pelo fato de sempre se produzir coisas novas, lida-se o tempo todo com o desconhecido e incertezas. É, portanto, imprevisível — basta notar o quanto de problemas brotam ao longo do desenvolvimento. E desperdício é fazer o mesmo software, do mesmo jeito e que gera o mesmo resultado — por isso que plataforma de produtos cabe tão bem na estratégia das empresas.

Apesar de opostas, a mentalidade de gestão de desenvolvimento de software é como a da indústria de manufatura, por isso, a interpretação acerca de a *agilidade* ser sobre produzir mais em menos tempo. O conceito por trás das metodologias ágeis é a **agilidade em obter resultado**, o que parece ser uma referência a ser veloz na execução de muitas coisas. Mas, acredita que pode ser justamente o contrário? Se o objetivo é antecipar o resultado, muitas vezes isso significa fazer menos! Em outros termos, fazer somente o que precisa ser feito, o suficiente para atingir determinado resultado.

Quando se antecipa o resultado, os impactos negativos e "*os problemas podem ser identificados e ajustados cedo, o que contribui para aumentar a eficácia da decisão*" (VALE, 2020). Para isso, é necessário fazer boas escolhas, questionando se realmente é preciso fazer o que se está fazendo, descartando o que não contribui na geração de valor.

Você já reparou que, quando a validação da hipótese é demorada, mais suposições surgem durante esse tempo? Na visão do Alisson Vale, isso torna "*a direção adequada para o projeto menos confiável. Dessa forma, reduzir o tempo de validação das hipóteses sobre se você está construindo o produto certo será chave para definir o seu nível de eficácia*" (VALE, 2020).

No geral, os conceitos de agilidade são aplicados na etapa de *delivery*, porém, quero propor a aderir a mentalidade da *agilidade* no *discovery* também: usar o tempo com sabedoria, evitando

desperdício de trabalho nas coisas erradas, que não trazem respostas ao que se precisa descobrir ou em coisas que não apresentam riscos significativos.

Por exemplo, um *benchmark* para conhecer como outros players resolvem determinado problema pode economizar um tempo considerável. Quando o iPod foi criado, a Apple não precisou pesquisar se "*as pessoas escutarão música em lugares públicos usando fones de ouvido*" (RIES, 2012). A Sony já tinha respondido essa questão com o Walkman. A Apple precisou descobrir se as pessoas estavam dispostas a pagar pelo download de músicas.

Plataforma também precisa de discovery

Dependendo do número de apps plugados na plataforma e times usuários, o *discovery* pode se tornar complicado por uma razão aparentemente boba: agenda — afinal, os times têm compromisso com as próprias metas deles. Além disso, quando os times dos apps fazem mudanças que não funcionam bem, isso impacta determinado grupo de usuários; quando o mesmo acontece numa plataforma, impacta todos os apps derivados, o que impacta todos os usuários finais desses apps. Como o problema é bem maior, exige um *discovery* mais cuidadoso e, claro, mais tempo. Por esses motivos, é fácil o *discovery* de plataforma se tornar longo demais.

Quando as pessoas da alta e média gestão de engenharia percebem o surgimento de problema, já imaginam como resolvê-lo. Por ser um produto técnico, que elas entendem bem, acreditam que sabem o que precisa ser feito e até como deve ser feito, por isso, não precisa de *discovery* — na mente delas. Esse comportamento também se deve ao costume de tratar plataformas como projeto de TI e a herança do modelo de gestão da era industrial.

É verdade que algumas coisas não precisam de *discovery*. Por exemplo, ao configurar algo na AWS, basta ler a documentação e seguir os passos. Outras precisam de, pelo menos, uma exploração técnica, como um trecho de código de teste, uma "PoCzinha" para

obter confiança de que a solução é factível. E outras tantas precisam de *discovery*, justamente porque o impacto pode ser avassalador para a organização.

Mas as pessoas tendem a ver plataformas da mesma maneira como veem os apps, "se houver impacto negativo, será pequeno"; basta dar o famoso *rollback*. Plataformas são mais complicadas do que isso. Das duas uma: ou a mudança não pode impactar de forma alguma ou deve impactar a todos. No segundo caso, é preciso um plano muito bem orquestrado envolvendo todos os times usuários, pois é muito provável que os times também tenham que alterar alguma coisa nos apps — e isso tem seu custo.

E quando se constrói algo ineficaz? Plataformas exigem um investimento inicial que pode ser alto demais quando se coloca na balança que não há retorno a curto prazo. Então, imagine dedicar tamanho esforço e longo tempo no desenvolvimento de algo que ninguém usará ou que traz problemas para a operação da empresa. Pode até não atrapalhar a operação, mas certamente a plataforma jamais proporcionará a eficiência aos times usuários que os stakeholders almejam.

Sabemos que, por mais autonomia que o time tenha, não muda o fato de chegarem demandas em forma de solução. Isso não é um problema, desde que a pessoa PM tenha em mente que é responsabilidade dela investigar antes do time “sair fazendo”. De cara, não sabemos se a tal “ideia” resolve o problema. Nem mesmo se o problema é o que parece, ou se vale a pena resolvê-lo. Como saber qual é a solução que vai resolver o problema? Aliás, qual é, verdadeiramente, o problema?

Como queremos evitar a construção de algo inútil e que também seja eficaz em contextos variados, se chegar uma ideia de solução em vez de problema, você pode trabalhar de trás para frente: entender o problema que imaginam que a tal solução vai resolver,

em seguida, entender o resultado esperado quando ela for usada e o impacto no negócio — fiz isso centenas de vezes, inclusive para a nova Order Platform.

Mas como fazer *discovery* de algo tão técnico como plataformas?

11.3 Como fazer discovery de plataforma

Você lembra que a gestão de produtos nasceu no marketing há décadas? O *discovery* também tem como base a administração de marketing, não é novidade no mercado. Na gestão clássica de produto, depois de definido o público-alvo, as necessidades e o posicionamento de mercado, inicia-se o desenvolvimento do produto. Apesar de não receber o nome *discovery*, o processo é o mesmo que o do marketing, como pode-se ver na imagem adiante e neste trecho do mais absoluto livro da área, *Administração de Marketing*, de Philip Kotler e Kevin Keller (2018):

"*Para transpor a etapa do plano de negócios e chegar à de desenvolvimento do produto, é necessário um* ***estudo de pesquisa de mercado convincente em relação às necessidades e aos interesses dos consumidores, uma análise competitiva e uma avaliação técnica****. A alta gerência analisa os critérios em cada estágio de controle para tomar uma de quatro decisões possíveis:* ***seguir adiante, abandonar, aguardar ou reciclar***." (grifos da autora)

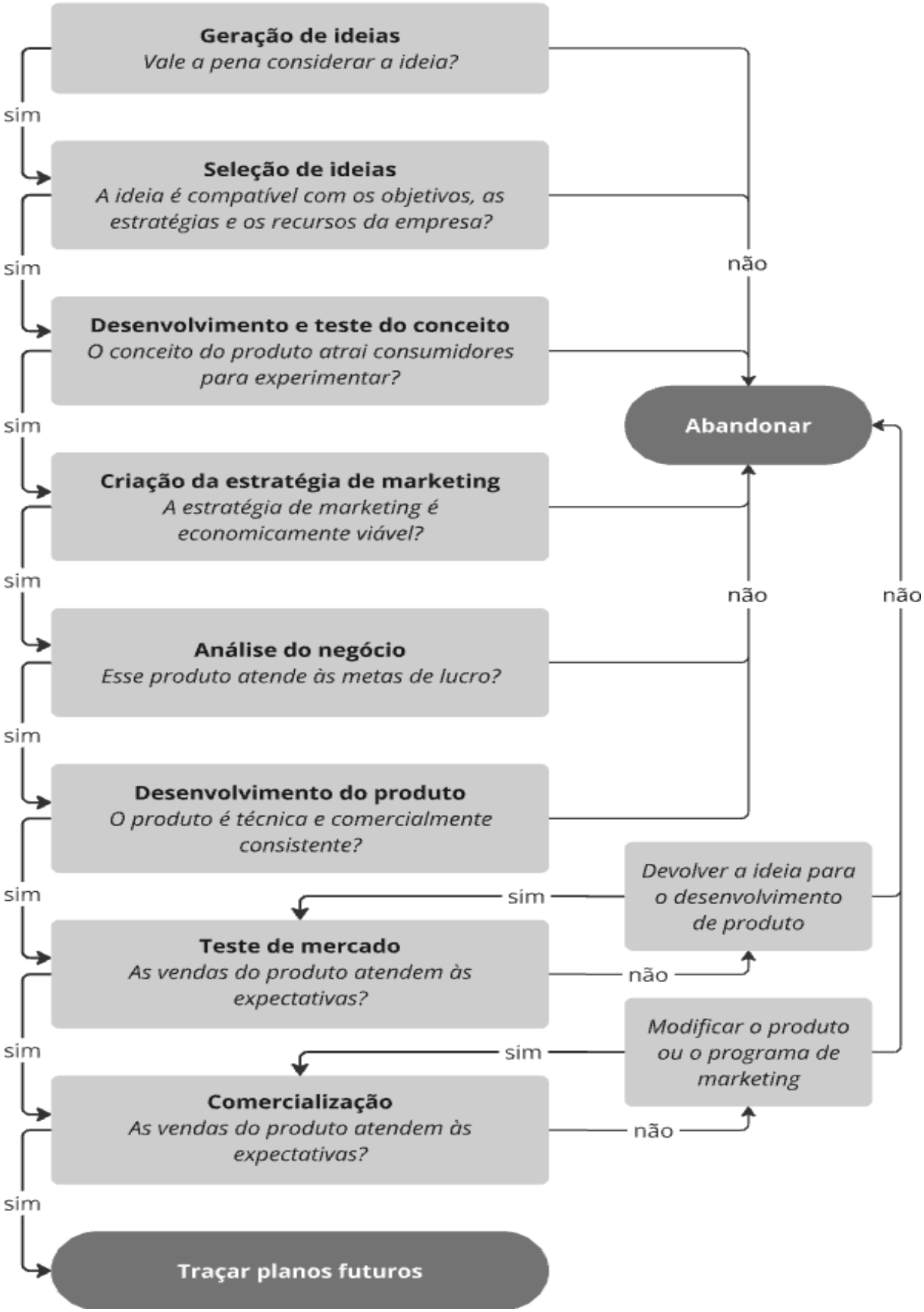

Geração de ideias
Vale a pena considerar a ideia?
sim
Seleção de ideias
A ideia é compatível com os objetivos, as estratégias e os recursos da empresa?
sim
não
Desenvolvimento e teste do conceito
O conceito do produto atrai consumidores para experimentar?
sim
Abandonar
Criação da estratégia de marketing
A estratégia de marketing é economicamente viável?
sim
não
não
Análise do negócio
Esse produto atende às metas de lucro?
sim
Desenvolvimento do produto
O produto é técnica e comercialmente consistente?
Devolver a ideia para o desenvolvimento de produto
sim
sim
Teste de mercado
As vendas do produto atendem às expectativas?
não
Modificar o produto ou o programa de marketing
sim
sim
Comercialização
As vendas do produto atendem às expectativas?
não
sim
Traçar planos futuros

Figura 11.1: Processo de decisão do desenvolvimento de novos produtos. Fonte: Administração de Marketing (KOTLER; KELLER, 2018).

A parte grifada no final da citação é o que conhecemos como **pivotar**. Além do famoso "pivô", popularizado com a publicação do livro *A Startup Enxuta*, Eric Ries (2012) traz que "*(...) precisamos de um método para decompor sistematicamente um plano de negócios nas suas partes componentes e testar cada parte de forma empírica. Em outras palavras, precisamos do método científico*". Agora, compare com o que traz o livro de marketing *Gestão e Desenvolvimento de Produtos e Marcas*, quando diz que os estudos de mercado e implementação são atividades que demandam "*administração científica, com métodos e técnicas*" (IRIGARAY *et al.*, 2011).

Viu como o método tradicional para desenvolvimento de produtos físicos é a mesma coisa do que conhecemos para produtos digitais? Quando percebi isso, pensei que qualquer tipo de produto poderia passar pelo processo de *discovery* — não seria diferente para plataformas.

Há uma infinidade de ferramentas para *discovery* e, com um pouco de criatividade, é possível usá-las para plataforma também. Para nos guiar à descoberta da plataforma eficaz — desejável, viável, factível —, usaremos o *Design Thinking* por ser uma ferramenta genérica, base de inúmeros *frameworks* que, provavelmente, você já conheceu em algum momento como PM.

Design Thinking para plataforma

A metodologia do Design Thinking é uma abordagem **centrada no ser humano** (*human-centered*) "*para integrar as necessidades das pessoas, as possibilidades da tecnologia e os requisitos para o sucesso dos negócios*"; em outras palavras, "*reúne o que é desejável do ponto de vista humano com o que é tecnologicamente viável e economicamente viável*" (IDEO, c2023) — não por coincidência, tem relação direta com os riscos que Marty Cagan aponta.

Os assuntos em torno de plataformas costumam ser tão técnicos que, às vezes, parece que estamos lidando apenas com máquinas e esquecemos que são pessoas que vão usá-las quando forem conectar seus apps. Os exercícios propostos pelo Design Thinking ajudam a gente a se concentrar no lado humano do desenvolvimento da plataforma — para ser eficaz, fazer a coisa certa, é preciso, antes de mais nada, envolver as pessoas certas, por isso os usuários têm papel chave.

Existem inúmeros frameworks que têm o Design Thinking como base. Aqui, usaremos o mesmo modelo da IDEO, proposto pelo Hasso Plattner Institute of Design da Stanford — também conhecido como *d.school*. Nesse modelo, o processo é dividido em cinco estágios:

1. Empatia;
2. Definição;
3. Ideação;
4. Prototipação;
5. Teste.

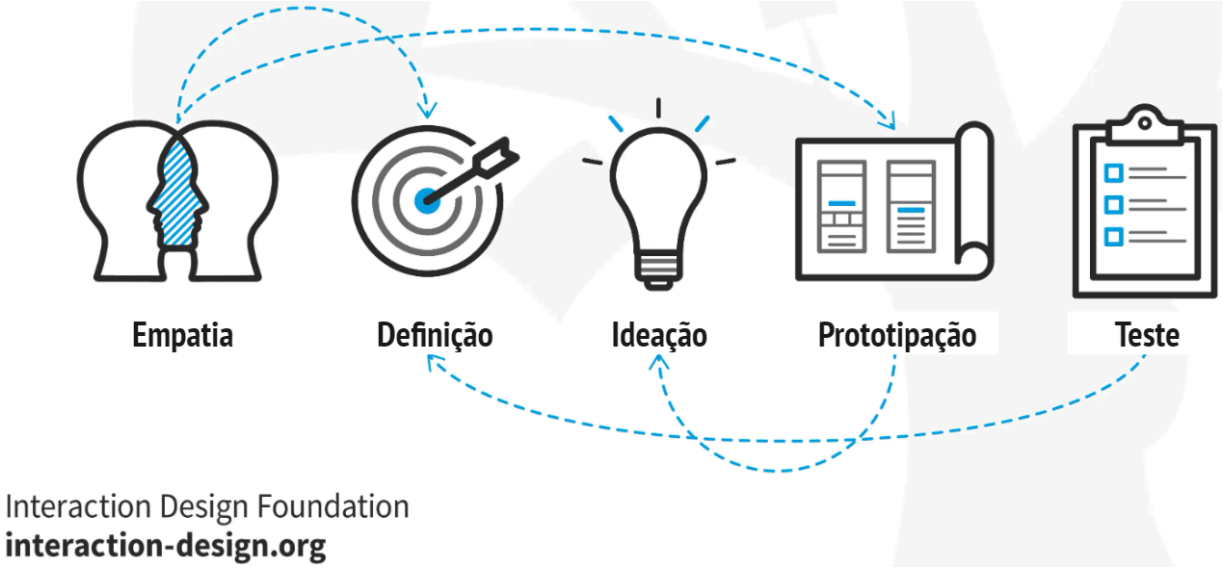


Figura 11.2: Os 5 estágios do Design Thinking. Fonte: Interaction Design Foundation (c2023).

Esses estágios nem sempre são sequenciais e algumas atividades podem caminhar em paralelo. Pela minha experiência, **nem é necessário passar por todas eles**. É comum identificar o problema e passar diretamente para a prototipação ou até mesmo para o desenvolvimento, por exemplo. O que importa é diminuir o tempo do ciclo de aprendizado: o período entre o levantamento da hipótese e a validação da solução com os usuários.

Além de não linear, o Design Thinking é iterativo. Portanto, tão comum quanto pular etapas é voltar etapas em qualquer momento do processo. Podem-se perceber *gaps* no protótipo antes mesmo de testá-lo e ser necessário voltar para a ideação. No teste, aprende-se mais com e sobre os usuários, o que pode levar de volta para a fase de empatia, ou surgirem novas ideias e voltar para a ideação. Os testes também podem revelar *insights* que redefinem o problema.

Para cada estágio, há uma infinidade de exercícios. A escolha de qual empregar deve ser feita com base em qual deles pode trazer respostas ao que se precisa descobrir e clareza ao que está nebuloso. É preciso manter em mente que o objetivo é entender os usuários em certo nível de profundidade e experimentar com segurança a fim de encontrar a solução na qual vale a pena investir tempo e recursos.

> A IDEO disponibiliza um repositório gratuito com métodos para a prática de Design Thinking e casos de estudo em http://www.designkit.org/.
>
> O Design Sprint tem algumas práticas que também podem ajudar: https://designsprintkit.withgoogle.com/.

A partir daqui, vamos passar por cada uma das cinco etapas, ver como usar ferramentas já conhecidas para descoberta de produtos *customer-facing* e vamos colocar a mão na massa para descoberta de plataformas, começando, no próximo capítulo, pela prática da *empatia* pelos times usuários.

Ah, uma coisa importante: o Lean Canvas é um documento vivo, então, ao longo da execução da descoberta, lembre-se de revisitá-lo e substituir as *hipóteses* por *fatos*.

Capítulo 12
Empatia: Exploração das necessidades

Uma prática centrada no ser humano não poderia começar sem compreender as pessoas a quem a solução se destina. O primeiro passo do Design Thinking é aproximar-se das pessoas que enfrentam os problemas para estabelecer **empatia**: conhecê-las de perto, em seus contextos, entender suas necessidades e desejos, como se comportam, seus pensamentos, emoções e motivações. Tudo isso ajuda a obter uma visão real dos usuários e afastar nossas próprias suposições, ao colocar luz sobre a maneira como eles fazem as coisas, o porquê e o que é significativo para eles.

A empatia é o alicerce do Design Thinking, crucial para plataformas, que costumam ser orientadas a tecnologias em vez de pessoas. Antes de colocá-la em prática, é preciso saber com quem empatizar.

12.1 Com quem empatizar

Quando segmentamos os clientes da plataforma no Lean Canvas, chegamos a uma visualização dos grupos com que precisamos estabelecer empatia. Vamos relembrar:

- **Empresa**, geralmente representada pelo CTO ou por alguém da alta gestão;
- **Times dos apps** (tech lead, PM e, em especial, devs);
- **Usuários finais** aos quais esses times atendem e a plataforma atende indiretamente.

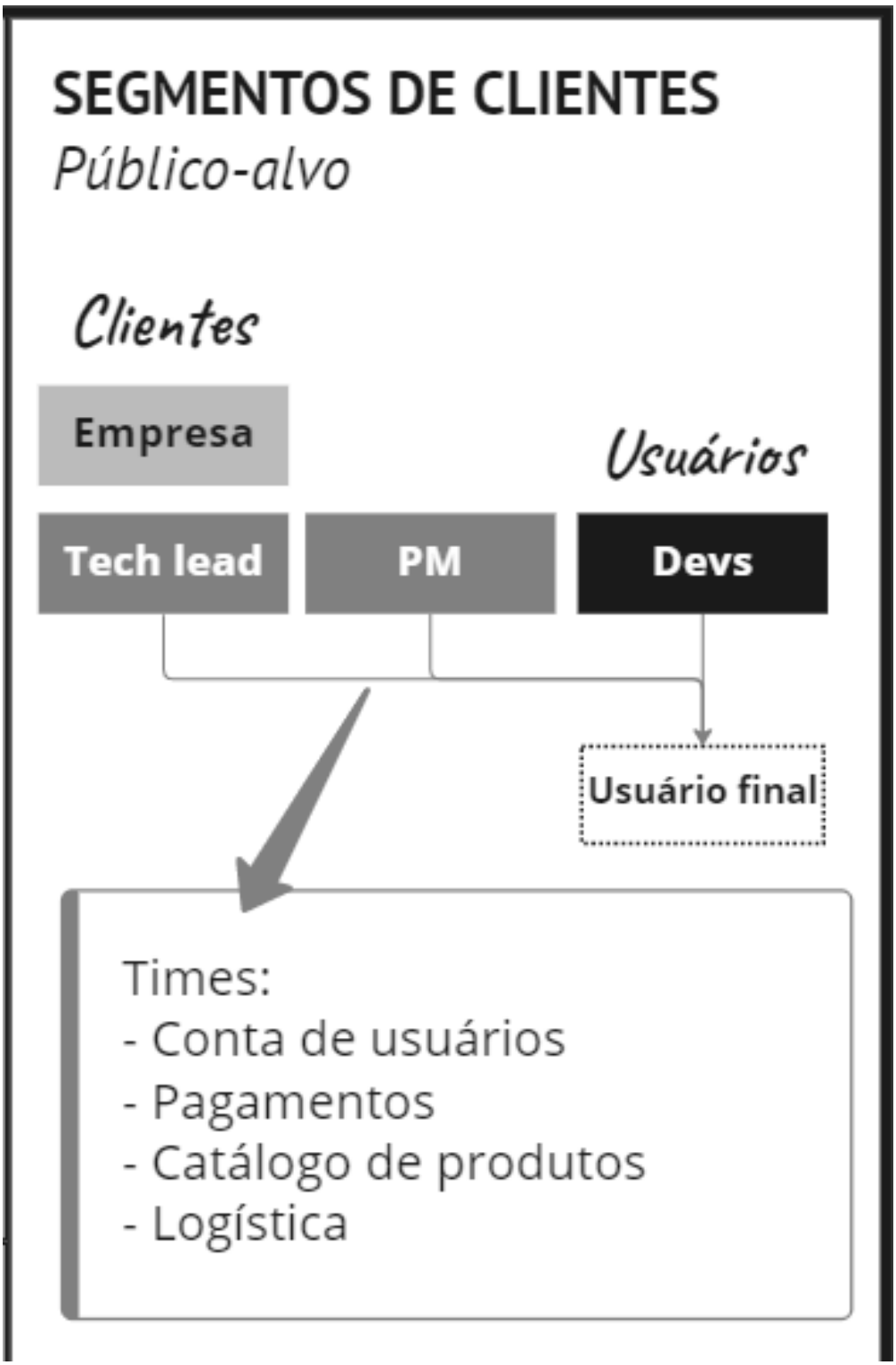


Figura 12.1: Lean Canvas — Segmento de clientes.

É importantíssimo aprender com os times usuários da plataforma o que eles sabem sobre os usuários finais. Acredite, quanto mais você souber sobre eles, mais informação você terá sobre o que incluir na API e como tornar mais fácil a experiência de uso do app (sim, do app, não das APIs). Então, além de entender a jornada dos devs que usam as APIs, devemos conhecer a jornada dos usuários finais para saber de que maneira a plataforma impacta a experiência deles. Dessa forma, conseguimos mais insumos para definir o que fazer em termos de *features* e o porquê — isso ajudará os times usuários a criarem melhores experiências para o usuário final.

Muito do *discovery* diz respeito a questões técnicas, como o estilo da API, dados a serem transmitidos, tipo de autenticação, entre outros. As escolhas das soluções dependem do público técnico que consome as APIs, e a chave para tomar essas decisões é conhecer quem vai usá-las e como serão criados os apps para os usuários finais.

Algumas questões que costumam surgir nessa fase para cada um desses segmentos são:

- Clientes da plataforma: Para quais times estamos fazendo isso? O que esses times fazem? O que estão querendo criar? Quais problemas estão enfrentando? Há alguma limitação da plataforma que torna a funcionalidade no app ineficaz para o usuário final ou que impede os times de usarem a plataforma?
- Usuários finais: Qual problema esses times querem resolver para os usuários finais? Por que os usuários finais estão passando por esse problema? Qual métrica demonstra o problema e qual a meta que esses times pretendem alcançar depois que o problema for solucionado?
- Dev experience: O que causa atrito no uso das APIs? Qual linguagem de programação os devs usam? Há bugs ou limitações na plataforma que impedem devs de implementarem o que desejam? Há alguma tecnologia que os devs precisam conhecer para usar as APIs?

O que você precisa conhecer dos clientes depende do seu momento no time e em qual estágio a plataforma está no ciclo de vida. Quando você começa em um novo time, é importante conhecer os clientes que já usam as APIs e as expectativas do *sponsor*. Quando o *discovery* é para uma nova versão da plataforma e você já conhece os clientes, a descoberta gira em torno do que já existe, o que pode ser melhorado e coisas novas que podem ser criadas.

Antes de partir para o *discovery*, é importante delimitar o que precisa ser descoberto. A Matriz CSD ajuda a organizar as informações e a exteriorizar a visão que cada pessoa do time tem em mente.

12.2 Matriz CSD para delimitar o discovery

A **Matriz de Certezas, Suposições e Dúvidas** — conhecida como **Matriz CSD** — é uma forma simples de trazer à tona certezas, suposições e dúvidas, como o próprio nome sugere. Ela atua como ponto de partida e alinhamento entre as pessoas do time.

É comum que product designers conduzam esse tipo de dinâmica, mas e quando não há PD no time? Nesse caso, você pode ser a pessoa facilitadora dessa e das demais dinâmicas que vamos explorar nesta parte do livro. Aqui está sendo considerado que você não foi PD no passado, por isso, vamos conhecer o passo a passo de como essa matriz funciona.

Passo 1: Delimitação do tema

Restringir o tema da discussão é a primeira coisa a ser feita. Do contrário, é fácil a dinâmica avançar em direções aleatórias e ficar longa demais. Quando há vários temas a serem descobertos, vale a

pena separá-los. Dessa forma, é possível andar com os temas em paralelo, se necessário.

Passo 2: Preparação para a dinâmica

Você vai precisar de algum meio para organizar as informações. Se a sessão for presencial, basta dividir um quadro branco (ou uma parede) em três colunas e nomear cada uma, como mostra a imagem a seguir. Se for remota, você pode escolher uma entre inúmeras ferramentas de *whiteboard*, como Miro, Mural, Microsoft Whiteboard ou Metro Retro.

Figura 12.2: Matriz CSD no Miro.

Isso também pode ser feito em formato texto no Google Docs ou Notion, por exemplo, e até em planilha no Google Sheets ou Microsoft Excel. O que importa são as informações e a organização delas. Usar diferentes cores ajuda a diferenciar, e nada impede que a visualização seja horizontal em vez de vertical (a mais comumente usada).

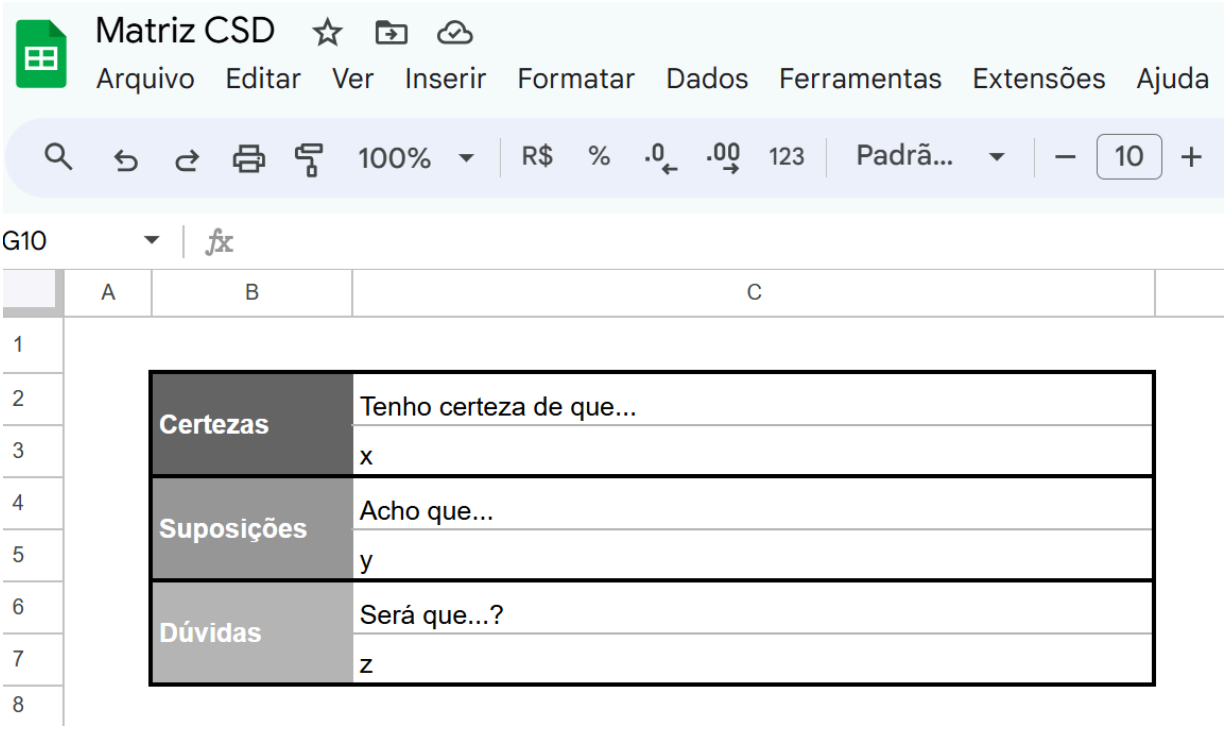


Figura 12.3: Matriz CSD no Google Sheets.

Nas áreas delimitadas, os participantes da dinâmica vão inserir as informações correspondentes:

- **Certezas**: os itens que os participantes da dinâmica já sabem;
- **Suposições**: os itens que se imagina que são verdades (mas dos quais não se tem certeza, portanto precisam de confirmação) e hipóteses de solução;
- **Dúvidas**: o que vem à mente em forma de pergunta.

O objetivo da matriz é organizar visualmente as suposições e dúvidas. À medida que se obtêm respostas, elas transitam para certezas.

Passo 3: Sessão com o time

O ideal é que todas as pessoas do seu time participem. Se for difícil de encontrar um horário livre em comum, o levantamento dos itens pode ser assíncrono e vocês se reúnem para comentá-los — também servirá para que todos fiquem na mesma página.

No início da sessão, explique o objetivo da dinâmica, em que a matriz vai ajudar e o que as pessoas devem preencher em cada quadrante. Se alguém do time nunca participou desse tipo de dinâmica, comece você o preenchimento para exemplificar como funciona: a característica dos itens e adicione um item para certeza, outro para suposição e mais um para dúvidas.

Além de facilitar a sessão, é importante que você também contribua com suas certezas, suposições e dúvidas. Se você já apurou dados relevantes para o tema, mostre-os antes que as pessoas comecem a popular a matriz, para fazer um alinhamento inicial e eliminar o desperdício de tempo em discussões cujas informações já foram obtidas.

Determine um tempo para o preenchimento da matriz. Não existe um tempo fixo: ele vai variar de acordo com a complexidade do tema. Pode começar com 10 minutos e aumentar se as pessoas precisarem de mais tempo. O tempo serve para manter o foco. Se sobrar tempo demais, as pessoas podem chegar ao ponto de começarem a trazer coisas irrelevantes — elas acham que precisam usar todo o tempo preenchendo com alguma coisa. Não precisa. Se nada mais vier à mente depois de 2 minutos de sessão, a pessoa pode sair para tomar um cafezinho, sem problema.

Depois que o time popular a matriz, é hora de passar por cada item. Você ou alguém do time lê em voz alta e outra pessoa pode ir agrupando os semelhantes. Como pessoa facilitadora da dinâmica, tente trazer clareza ao que foi escrito. Pergunte, provoque e altere o texto depois que todos chegarem ao mesmo entendimento.

Por exemplo, vamos supor que alguém escreveu “os devs não sabem usar a API”. Isso é aberto demais. Por que os devs não sabem? O que demonstra isso? Essa informação fica específica mudando para “os devs confundem os atributos de pagamento de cartão com os de PIX”. Isso mostra que os devs sabem usar outras coisas, portanto, essas outras não são dores — ou são, mas por outro motivo.

Atenção redobrada às *certezas*! É praxe as pessoas verem ou ouvirem falar alguma coisa e considerarem uma certeza. Questione! Se a justificativa começar com "acho que…", "uma vez eu vi...", "ouvi falar que...", não é *certeza*, é *suposição*. Transfira o *card* (texto ou a linha da planilha) para o local correspondente e refraseie, se necessário. O ideal é manter como certeza somente o que se tem algum tipo de evidência: dados quantitativos, qualitativos (trechos de falas), documentação, *prints*, trechos de código, entre outros. É comum aparecerem dúvidas ou suposições que alguém do time tem como certeza, e elas podem ser sanadas com alguma evidência — por exemplo, abrir o código na hora.

Passo 4: Levantamento dos riscos

Depois que a visualização estiver organizada, analise, junto aos participantes, os pontos que apresentam maior risco. Aqui é onde podemos exagerar ao selecionar coisas demais, deixar o *discovery* longo e mais demorado do que o ideal. O *discovery* deve iniciar pelos riscos mais críticos, aqueles que podem impedir ou limitar o sucesso da plataforma. Se estes se confirmarem, de nada adianta descobrir os demais, porque o sucesso não tem chance de acontecer.

Relembrando o que foi dito no capítulo anterior: a intenção é *diminuir riscos*, não eliminá-los por completo. Tentar eliminá-los toma tempo demais e a ideia pode se tornar obsoleta rapidamente, já que as tecnologias avançam vertiginosamente.

Uma vez selecionados os pontos nos quais se investirá tempo, isto é, que entrarão para o objetivo do *discovery*, é hora de estabelecer empatia. Para isso, você pode recorrer a técnicas que, como PM, você provavelmente já conhece: análise de dados, entrevista com usuários, pesquisa por formulários e observação.

12.3 Análise de dados

Se você já atuou como PM de produto *customer-facing*, você deve ter usado testes A/B e muitas outras métricas para orientar suas decisões. No entanto, ao assumir o papel de PM de plataforma, você deve ter percebido que é uma realidade completamente diferente. Algumas dezenas de times usuários não fornecem volume de dados que tragam confiança estatística na conclusão do teste.

Na Parte 2, passamos pelo capítulo de métricas e vimos que elas ajudam a entender certos problemas sobre tecnologia, mas deixam muitas dúvidas e suposições em aberto quanto a funcionalidades. Uma abordagem alternativa seria utilizar as métricas dos times de usuários para analisar os dados. Por exemplo, alto índice de abandono em uma determinada etapa da jornada do usuário pode sugerir a necessidade de ajustes na API para permitir ou impedir determinadas ações por parte do usuário.

12.4 Entrevista com clientes e usuários

A entrevista com os times usuários pode ser feita por qualquer meio, até mesmo em forma de texto pelo Slack ou WhatsApp, por exemplo. Para isso, com base na Matriz CSD, elabore as perguntas de modo que permita que os entrevistados até desviem para caminhos inesperados. Dessa forma, você descobre os *unknown unknowns* — coisas que não estavam no seu radar porque você nem sabe que não sabe e, por conta disso, podem trazer riscos à solução. Só tenha cuidado para garantir que você consiga as respostas que precisa para a descoberta. É importante saber o porquê das coisas para entendê-las mais profundamente.

Se você não conhece o termo *unknown unknowns*, veja na Wikipedia (em inglês): https://en.wikipedia.org/wiki/There_are_unknown_unknowns

Na prática, a elaboração das questões é como a que fazemos com produtos *customer-facing*: perguntas abertas e, eventualmente, perguntas fechadas (sim/não) que levam à próxima questão aberta. Por exemplo, na migração para a nova versão da Order Platform, a preocupação era se os times já usavam Kafka. Então, uma das perguntas era: *Kafka é um problema para vocês?* Se a resposta fosse *não*, a entrevista seguia para o próximo tema. Se fosse *sim*, a próxima pergunta era relacionada a quais seriam as dificuldades, porque tínhamos dúvidas se um tutorial de como fazer seria suficiente para os times usuários ou se seria necessário que uma pessoa dev do time da Order Platform ficasse temporariamente como "consultora" para ajudar na implementação.

12.5 Pesquisa por formulários

Às vezes, queremos saber exatamente a mesma coisa de diferentes times. Se as perguntas não exigirem grandes explicações nem histórias nas respostas, **formulários** podem ser a melhor alternativa.

A Order Platform fornece a API para busca de pedidos, a Order Search. No desenvolvimento da nova versão da plataforma, tínhamos dúvidas se tudo era realmente usado, e já conhecíamos alguns usos indevidos — sabe como é, tem usuário que usa a solução de forma para as quais ela não foi idealizada. Na época em que essa API foi criada, ela não foi elaborada exigindo autenticação, então não conhecíamos todos os times usuários e para que usavam

a API. Por isso, elaboramos um formulário para descobrir essas informações.

A seguir, um exemplo da documentação, baseada em história real — não são os fatos em si: informações foram modificadas ou retiradas para não comprometer a confidencialidade e a ética.

Caso baseado em uma história real

Primeiro, a descoberta foi delimitada à Order Search e, como era algo "pequeno" e já tínhamos um problema claro, foi criado um documento no Google Docs que iniciava pelas **certezas**.

> **O QUE SABEMOS**
>
> A Order Search foi herdada do serviço de *back office* e novas formas de buscas foram adicionadas, mantendo a base que já existia:
>
> - Busca por ID;
> - Busca filtrada por atributos do pedido;
> - Busca por status do pedido;
> - Busca filtrada por valores agregados.
>
> Limitações:
>
> - Disponibiliza somente o dado atual;
> - O ElasticSearch oferece pouca margem para customizações por ser gerenciado.

Veja que a tecnologia (ElasticSearch) era uma preocupação e havia dúvidas sobre se deveríamos mudar para outra. Assim, o *discovery* foi dividido em dois: *técnico* e *com usuários*.

Discovery técnico

O *discovery* visa diminuir incertezas sobre "o quê" e "como" devem ser as APIs da nova plataforma. Para isso, precisamos de algumas informações antes de seguir para a entrevista com os usuários. Assim, a análise técnica da Order Search deve responder:

- Qual o *throughput* atual?
- Quais filtros são usados de fato?
- Qual o período máximo das buscas?
- O ElasticSearch oferece flexibilidade para qualquer tipo de busca? Continua sendo uma boa opção ou tem algo melhor para suportar o crescimento do iFood?

A análise dos dados e a análise técnica do Elasticsearch (e de alternativas que poderiam substituir essa tecnologia) eram atividades a serem executadas pelo próprio time. Outras informações precisavam de respostas dos usuários e a escolha foi buscá-las via formulário. A entrevista ocorreria se houvesse necessidade.

Discovery com usuários

Não conhecemos todos os times que usam a API Order Search. Precisamos saber quem usa, para que usa e se há necessidade de agregar os dados. Para isso, será compartilhado um formulário para que os usuários respondam às seguintes questões:

1) Qual é o nome do seu time?

2) Quais features da Order Search seu time usa e qual o período de dados é necessário para cada uso?

- Busca por ID:
  - Menos de 1 mês
  - 1 mês
  - 2 meses
  - 3 meses
  - Mais de 3 meses
  - Não usa

A questão 2 foi repetida para todos os outros tipos de buscas.

3) Para que o seu time usa essas features? Qual o valor desses dados para os usuários da sua aplicação? Quais problemas podem ocorrer caso essas features deixem de existir?

4) Tem alguma finalidade para a qual você gostaria de utilizar a Order Search e não está disponível? Para o que ela seria necessária? Que alternativa você está usando?

O time da Order Platform contribuiu diretamente no documento com o que queriam saber. Depois disso, foi criado o formulário no Google Forms e a "convocação" para o preenchimento foi via Slack.

Para garantir que as perguntas levariam às respostas que precisávamos (e, se não trouxessem, seriam necessárias outras

formas de consegui-las), adicionei o que era esperado desse *discovery*.

Resultado esperado do discovery

O *discovery* deve ajudar em decisões, como:

- O que deve ser fornecido? (Lista com os usos, buscas, filtros)
- É necessário TTL?
- O ElasticSearch suporta o crescimento previsto? Se não, qual tecnologia usaremos?

Às vezes, análise dos dados, entrevista e formulário de pesquisa não são formas ideais ou suficientes para identificar problemas. Neste caso, conseguimos entender melhor por meio de *observação*, acompanhando o que está acontecendo junto com o dev usuário.

12.6 Observação

Observar é uma maneira de entender melhor o contexto, a jornada, como devs interagem com as APIs e documentações, como fazem as coisas e o porquê. Ajuda a identificar o que os usuários precisam. Também mostra se o que a pessoa diz corresponde ao que ela faz: a pessoa pode fazer coisas e esquecer de mencioná-las em uma conversa ou considerá-las irrelevantes, mas que a gente consegue perceber ao observar.

Esse método pode ser combinado com a entrevista. Repare como a pessoa dev conclui determinada tarefa e peça para que explique por que ela está fazendo o que faz. Peça para que fale em voz alta o que está passando pela mente dela enquanto realiza a tarefa. Procure entender o contexto do time usuário, as histórias por trás dos

artefatos que eles criam e acessam. Conheça os ambientes e ferramentas que eles usam para fazer perguntas mais assertivas.

Peça para que mostrem onde e o momento em que ocorrem problemas. Assim você terá uma compreensão mais completa e pessoal do que pode estar por trás das experiências e motivações. Se o problema for técnico, alguma pessoa de engenharia do seu time pode ajudar você a identificá-lo e "traduzi-lo" de modo que você entenda.

12.7 Cuidados

A empatia é útil para obter uma visão real dos usuários, mas é importante se atentar às suposições que o usuário tem. Você pode cair em uma armadilha ao acreditar que demandas que chegam por pessoas de engenharia não precisam de investigação.

É verdade que, como os usuários são pessoas técnicas, é possível que eles vislumbrem a potencial solução com assertividade — geralmente, eles conseguem identificar os problemas e anormalidades pelos logs ou pelo código. Por outro lado, elas não conhecem a plataforma no mesmo nível que o time que a desenvolve, tampouco as necessidades de outros times. O valor de uma plataforma está em fornecer um sistema útil a diferentes contextos, e os devs usuários não sabem se a solução que estão propondo é a melhor para todos eles.

Também é preciso evitar ir no sentido oposto, isto é, investigar demais um problema que é óbvio ou cuja solução é tão simples e sem risco para o funcionamento da plataforma e do ecossistema que vale mais a pena partir direto para a implementação.

Plataformas devem resolver problemas de grande escala, então também devemos avaliar o risco de não resolver o problema. Se o risco for baixo, é provável que seja um problema que não vale a

pena ser resolvido — pelo menos não na plataforma. Talvez seja mais coerente resolvê-lo diretamente no app que sofre o problema.

Agora, supondo que o problema seja típico para ser resolvido na plataforma, tudo o que foi visto e ouvido na fase de empatia deve levar a conclusões de um quadro geral. É capaz que esta etapa resulte em informações não estruturadas, casos de uso/requisitos sobrepostos ou conflitantes. Geralmente, não são todas as pessoas do time que participam da coleta das informações, então é preciso sintetizar e compartilhar os achados com todo o time, o que leva à próxima fase, a de *Definição* — e como fazê-la é o assunto do próximo capítulo.

Capítulo 13
Definição: Entendimento dos problemas e necessidades

A etapa de *Definição* do Design Thinking visa trazer clareza e restringir o foco das próximas etapas. É o momento de juntar os dados brutos coletados na etapa de *Empatia*, separar o que pode ser realmente usado como informação, analisar, interpretar e perceber tendências a fim de identificar o problema.

A *Definição* molda tudo o que virá nas próximas fases, por isso é uma das mais importantes de todo o processo. Errar nesta fase causa desperdício de tempo e recursos na direção de algo que não é importante. Esta etapa pode ser trabalhosa e demorada, dependendo da quantidade de informação a se processar.

13.1 Análise e síntese das informações

As informações — *prints*, observações, citações, jornadas ou experiências — devem ser dispostas em algum lugar de modo que você consiga fazer conexões entre elas. Isso pode ser feito em uma parede, *whiteboard* ou até mesmo no Google Docs. Selecione apenas as informações relevantes, capazes de gerar *insights* e direcionamento, e deixe de fora o que for considerado irrelevante. A apresentação pode ser visual ou textual, desde que o conteúdo seja de fácil consumo.

Separe-as em grupos e classifique-as. Analise conexões, padrões e tendências que impactam diferentes usuários da plataforma e cenários. Tente definir casos de uso e requisitos a que você pretende atender. Essa etapa do processo também é o momento certo para

sinalizar dependências que a API possa ter com outras plataformas — comum em arquitetura de microsserviços.

Alguns métodos podem ser úteis para organizar e dar visibilidade: Diagramas de Afinidade, Mapas de Empatia, *Jobs to be Done* e *Customer Journey Maps* ajudam a destrinchar os problemas; documentação no formato *Opportunity Assessment* ou *Product Required Document (PRD)* para novas funcionalidades ou melhorias significativas; *Business Canvas Model* ou *Lean Canvas* para grandes iniciativas, novas plataformas ou novas versões (apresentados na Parte 2). Na prática, a sintetização combina diferentes ferramentas.

13.2 User story mapping

Para compreender a jornada dos desenvolvedores usuários na nova plataforma, apliquei o método *user story mapping* (mapeamento da história do usuário) — minha ferramenta favorita, em disparado. É uma técnica visual que ilustra a jornada ao interagir com o produto. Você pode organizar usando *post-its* na parede, em planilha ou em softwares de *whiteboard*. A seguir, o *print* da ferramenta no Miro.

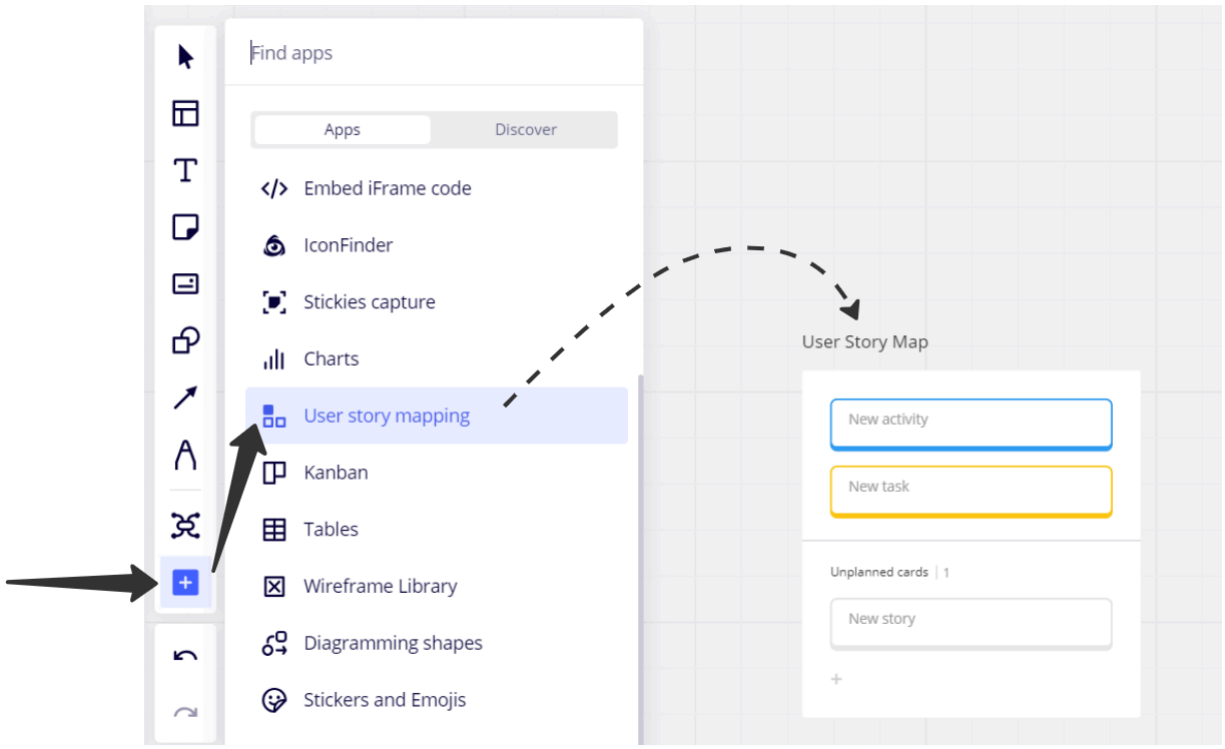


Figura 13.1: User Story Map no Miro.

Criado por Jeff Patton (2014), autor do livro *User Story Mapping*, o mapeamento é estruturado em três camadas — atividades, tarefas e detalhes —, formando uma "*grade simples que narra uma história da esquerda para a direita e a detalha de cima para baixo*" (tradução livre). Esta estrutura ajuda a definir o que será construído e como será a experiência do usuário.

A primeira camada, *atividade* (*activity*), organiza horizontalmente os objetivos do usuário ao usar o produto. Abaixo, a camada *tarefas* (*tasks*), apresenta os passos necessários para alcançar esses objetivos. Por fim, na terceira camada, *detalhes* (*details*), lista verticalmente as subtarefas do usuário e variações delas, aspectos técnicos e de UI — aparência da interface (*look and feel*) e o que os sistemas fazem em segundo plano. Esses detalhes são as *user stories* que vão para o backlog.

Figura 13.2: Camadas do User Story Mapping.

> Vale a pena dominar a técnica de *user story mapping*. Se for do seu interesse, a melhor recomendação é o livro escrito pelo Jeff Patton que leva o nome da ferramenta, *User Story Mapping*. Ele também publicou o post *The New User Story Backlog is a Map* (https://jpattonassociates.com/the-new-backlog/), onde ele explica a ferramenta de forma breve e disponibiliza alguns slides explicativos (https://jpattonassociates.com/storymappingslides/).

O mapa conta a história sobre determinado perfil de usuário que faz algo para atingir um objetivo. Na prática, a narrativa abrange a

história de mais usuários. Uma vez que o dev integra o app dele à API, o sistema conectado à plataforma atua como um usuário (sistema consumidor). No contexto de plataformas, é inevitável incluir interações entre sistemas na visualização, inclusive o próprio Patton diz que "*não tem problema antropomorfizar serviços de back-end ou coisas complexas que o sistema faz*". Os tipos de usuários e interações podem ser diferenciados por tags, cores e/ou ícones, conforme ilustrado na imagem a seguir.

Figura 13.3: Usuários.

Caso baseado em história real: Jornada dos devs

Para a nova Order Platform, organizei o *user story map* com o que eu conhecia sobre a jornada, e o time contribuiu refinando-o durante workshops em que definimos o que precisaríamos fazer para concretizar a jornada.

Começamos revisando cada atividade e tarefa da jornada. O time colaborou para preencher lacunas. Em seguida, detalhamos cada tarefa. Nesse processo, surgem aspectos em que não se havia pensado antes e dúvidas — basta adicionar um card na camada de *detalhes* do mapa com dúvidas, observações e lembretes, por exemplo.

A imagem a seguir mostra um pequeno trecho (simplificadíssimo) da jornada. A atividade "*aprende como integrar a Order Platform*" foi condensada em uma tarefa, "acessa a documentação". Abaixo dela, listamos o conteúdo a ser criado, independente de ser acessado em

diferentes páginas ou na mesma. O importante era tornar visível e para lembrar o que tínhamos em mente. Seguindo a abordagem "*just in time*", seriam quebrados em *user stories* "menores" no backlog no momento em que se tornassem relevantes.

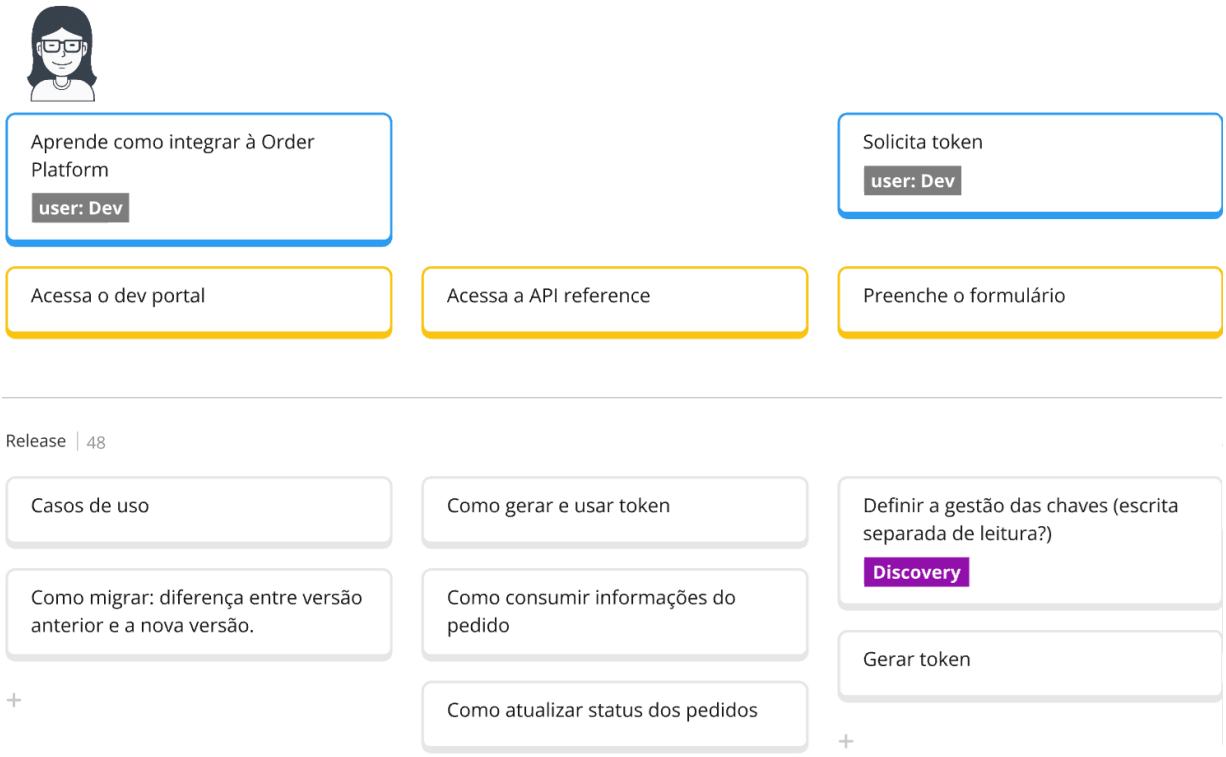


Figura 13.4: Trecho da jornada de uso da Order Platform.

Note o card com a tag "Discovery". Esta marcação indica que mais *cards* serão adicionados após a fase de descoberta, geralmente relacionados ao *delivery*.

O próximo trecho mostra a jornada sob a perspectiva de sistemas. A tag "Discovery" na primeira coluna refere-se à ideação da solução, que demandava a participação de diversos times, enquanto a última coluna destaca uma dúvida que seria necessário *discovery* técnico. Além do que precisaria ser executado pelo time, há pequenos comentários para trazer clareza e visibilidade.

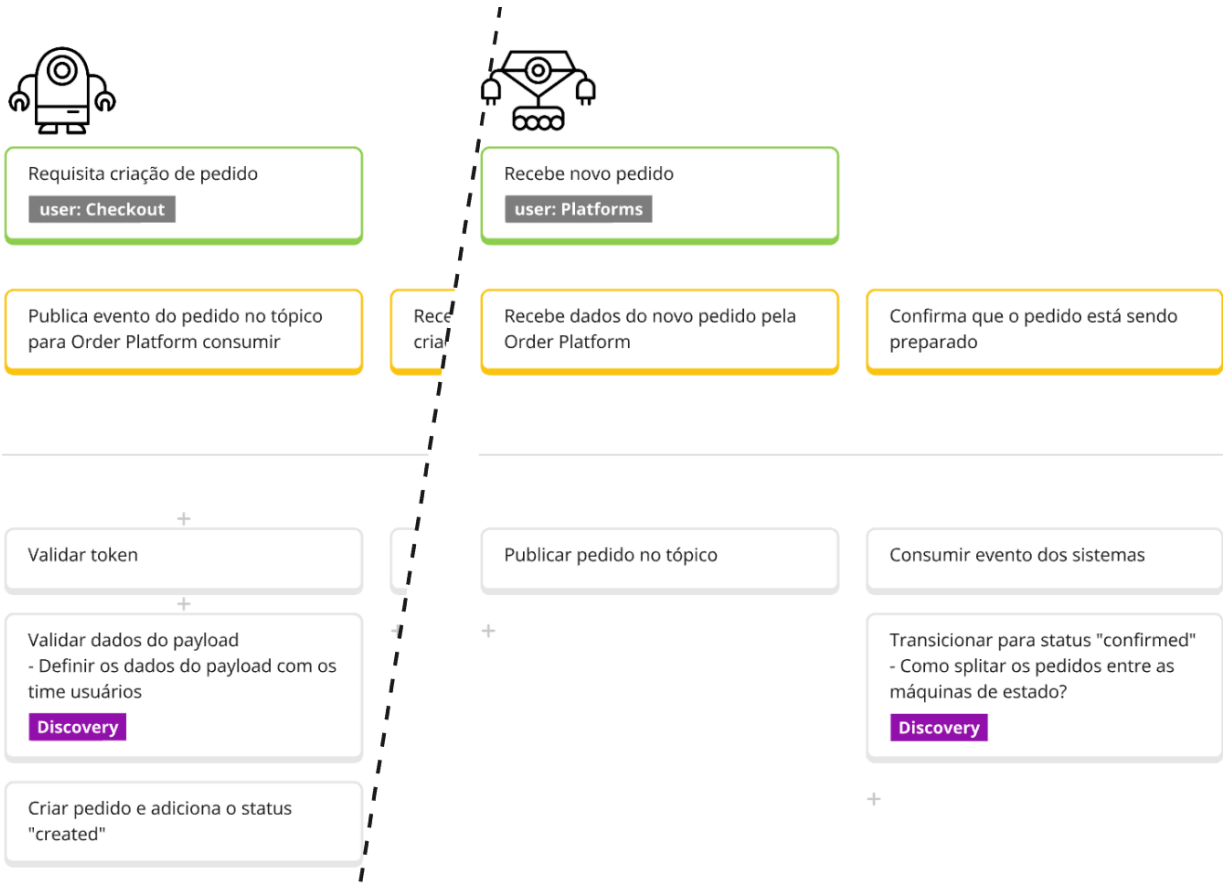


Figura 13.5: Trecho da jornada de uso da Order Platform.

A facilitação da dinâmica de *user story mapping* no primeiro time de plataforma que trabalhei no iFood foi um sucesso, mas no time da Order Platform encontrei resistência. O time acreditava que já sabia o que precisava ser feito e não via valor no processo. No entanto, a entrada de grande número de novos itens no backlog indicava o contrário.

Sempre que surgiam muitos itens novos, eu questionava o time, reforçando que as tarefas deveriam ter sido identificadas durante a dinâmica. As pessoas argumentavam que algumas tarefas emergiam durante o desenvolvimento (o que é normal), enquanto outras não eram consideradas importantes. Após essa constatação, o time começou a entender melhor o processo e as dinâmicas seguintes

fluíram melhor. Em um dos planejamentos, conseguimos mapear toda a jornada e as tarefas necessárias — ainda longe do ideal, mas infinitamente melhor do que nas primeiras sessões.

Caso baseado em história real: Jornada dos usuários finais

Uma das principais *features* da Order Platform é a gestão do ciclo de vida do pedido, que inicia na criação e perpassa por todos os status até ser entregue ao consumidor. Para *delivery* de refeição, a *feature* atendia bem, mas ao longo do tempo o iFood passou a fornecer outras formas de vender e entregar refeições, e somado a isso, produtos de mercado, farmácia, entre outros. Esses outros cenários possuem operações diferentes, por isso, o status de pedido não era totalmente coerente, o que levou a inúmeras gambiarras pela empresa. A missão da nova plataforma era oferecer uma máquina de estados (talvez mais de uma) que atendesse melhor a variados cenários, inclusive pensando no futuro, imaginando casos que nem saberíamos se um dia iriam existir.

A primeira coisa foi entender o que não ia bem para os usuários finais de acordo com o tipo de pedido. Entrevistamos os times (PMs junto de pelo menos uma pessoa de engenharia) para entender a jornada do usuário final. Para essas conversas, criei uma jornada simples com o que sabíamos e os respectivos status — se você já usou o app do iFood, você tem uma noção —, para que pudéssemos detalhar e identificar os pontos em que a plataforma não atendia bem.

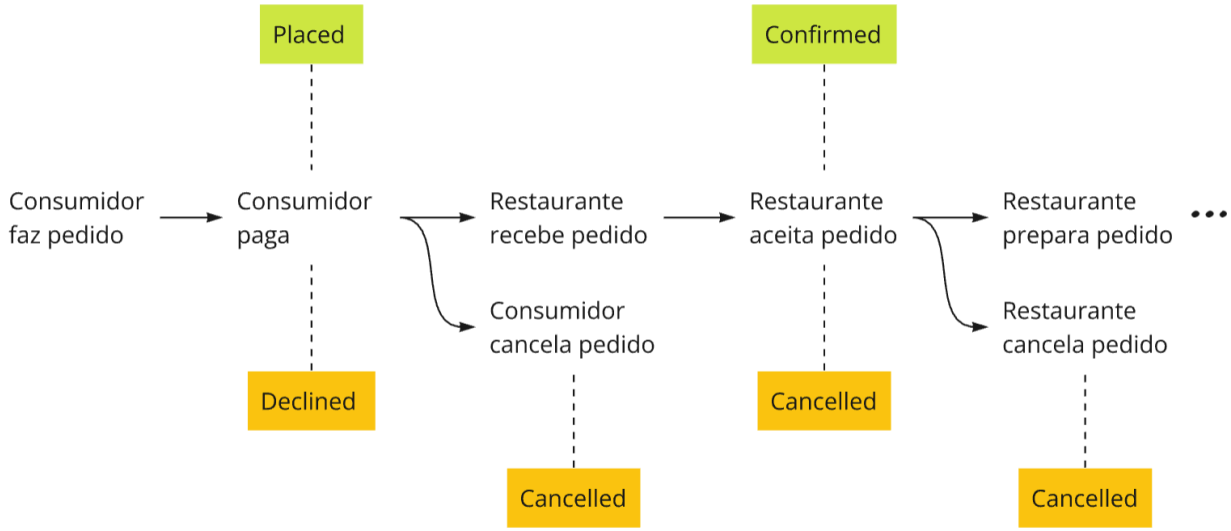


Figura 13.6: Jornada do usuário final para conduzir a entrevista.

Durante a conversa, incrementamos junto a pessoas do time entrevistado. Apareceram muitas coisas que aconteciam por trás e das quais não tínhamos noção — por exemplo, o envolvimento de outros sistemas e processos, marcados com estrela na figura a seguir. Comigo, participava pelo menos uma pessoa mais experiente de engenharia da Order Platform, que fazia o papel de tech lead.

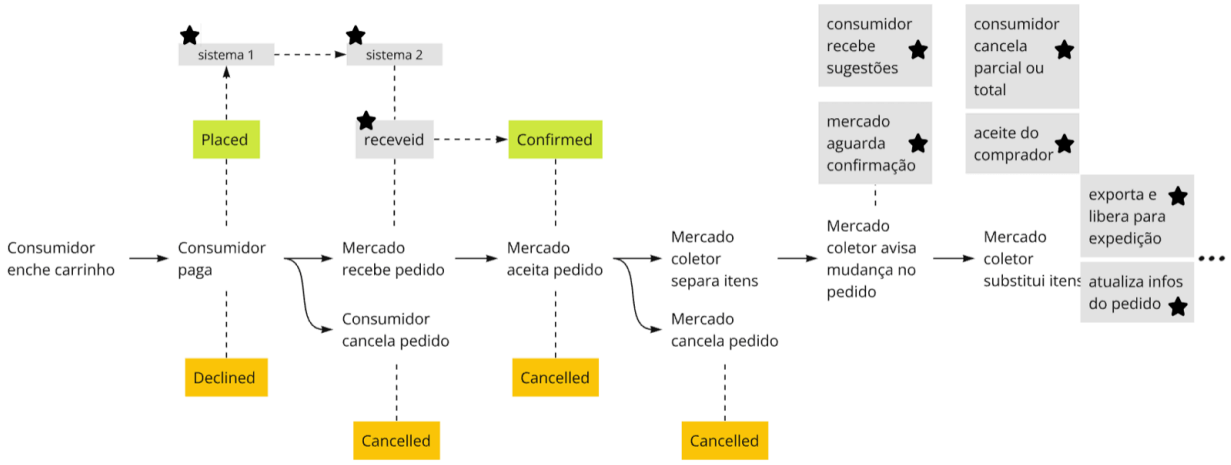


Figura 13.7: Jornada do usuário incrementada durante a entrevista.

A imagem adiante mostra um trecho de como organizei as informações para cada fluxo de pedido depois das entrevistas,

incluindo *features* da Order Platform e sistemas externos a ela, para compartilhar os aprendizados com time e mostrar onde havia deficiências, como status incompatíveis com a realidade da operação ou momentos do ciclo de vida a que não atendíamos.

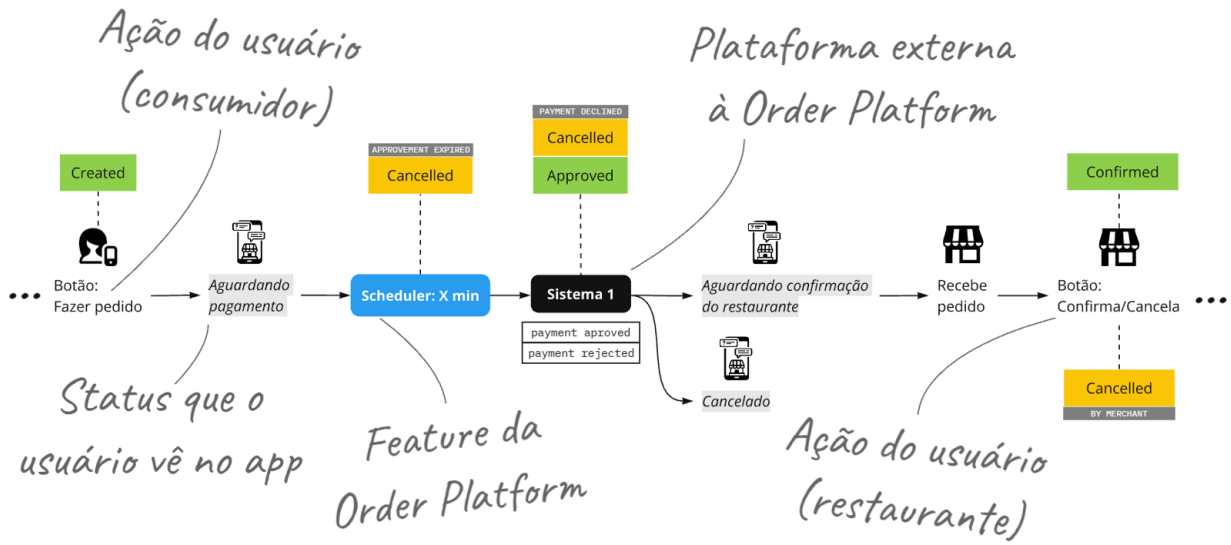


Figura 13.8: Jornada do usuário.

Esse material serviu de base para a próxima etapa do Design Thinking, a *Ideação*. Antes disso, durante o alinhamento de informações da jornada com o time, delimitamos os problemas por escrito.

13.3 Declaração do problema

Lembra que o suposto problema foi listado no Lean Canvas? A empatia e a sintetização das informações ajudam a refinar o problema descrito no início. Assim, ele se confirma, ou não. Talvez se revele mais extenso ou mais simples do que se imaginava, o que torna necessário reformular a descrição inicial em uma definição de problema mais clara e detalhada.

Também conhecida como **Ponto de Vista** (*Point of View* — POV), a declaração de problema guia o próximo estágio em direção à construção da coisa certa — a *eficaz* —, por isso, não pode ser vago ou aberto a interpretações variadas por diferentes pessoas ou contextos. Também não pode conter nada referente à implementação da solução — isso restringe a exploração de ideias inesperadas que podem trazer valor. O ideal é que a declaração do problema tenha foco limitado e critérios que ajudem a avaliar ideias concorrentes. Deve ser algo simples e conciso que, de acordo com Dan Harel e Lorraine Justice (c2019; tradução livre), professores do Rochester Institute of Technology, mostre:

- Deficiências, ineficiências ou incapacidade de resposta às necessidades na situação atual;
- O time ou entidade que vai se beneficiar da solução;
- Por que o problema é importante para os times usuários e/ou usuários finais;
- Por que as soluções anteriores não foram bem-sucedidas;
- As vantagens da nova solução;
- Oportunidades mais promissoras e os potenciais riscos.

A entrevista com um dos times usuários da Order Platform poderia se resumir na seguinte definição de problema (aqui é apenas um exemplo):

"*A funcionalidade X do app é subutilizada, porque os usuários finais não têm tempo hábil para interagir. Isso ocorre devido à regra de limite de tempo para interação, ideal para unidade de negócio A e inapropriada para a unidade B, C e D. Sem prejudicar a operação das demais unidades de negócios, a nova solução pretende fornecer a capacidade para o time XPTO de aproveitar o tempo ocioso entre a criação do pedido e a preparação, para que os usuários possam interagir com a funcionalidade e, dessa forma, evitar cancelamento de pedidos com objetivo de aumento de rentabilidade.*"

Uma vez que a definição do problema é estruturada e conclui-se que resolvê-lo vai ao encontro dos objetivos de negócio e que vale a

pena investir tempo e esforços na solução, parte-se para exploração de possíveis soluções, a *Ideação*. É o que faremos no próximo capítulo.

Capítulo 14
Ideação: Levantamento de ideias de soluções

Problema identificado e bem delimitado, é o momento de encontrar alternativas para solucioná-lo considerando diferentes perspectivas. O estágio de *Ideação* se concentra na geração de ideias e as mais promissoras fornecerão material para construir protótipos a serem testados pelos usuários, e o *feedback* deles dirá se é a melhor solução ou não.

Existem várias técnicas de ideação. *Brainstorm*, *How might we...?*, Mapa Mental (*mindmapping*) são tipicamente usadas no início desse estágio, e *Wireframe* é uma forma de começar a concretizar a ideia. Alguns métodos podem ser realizados individualmente; outros, precisam de colaboração entre pessoas do time da plataforma, times usuários e/ou outros *stakeholders*.

14.1 How might we...?

Se você não sabe por onde começar, o método "*How might we...?*" (HMW) — que significa "Como podemos...?" ou "De que maneira podemos...?" — é uma ótima maneira de instigar a geração de ideias, por isso é comumente usado para abrir sessões de ideação.

Uma questão HMW devidamente enquadrada não traz consigo uma solução específica, apenas oferece a estrutura para pensar. Dessa forma, pode ser respondida de várias maneiras.

A dinâmica começa pela reformulação da declaração do problema em forma de pergunta. Informações coletadas no estágio de *Empatia*, assim como outras que possam ter surgido até esse momento, servem para formular demais questões. Então, para um

problema Y, em vez de dizer "*precisamos projetar X*", se diz "*Como podemos evitar que Y aconteça?*".

Por exemplo, a declaração apresentada no capítulo anterior — "*(...) Isso ocorre devido à regra de limite de tempo para interação (...)*" — poderia ser reformulada com foco no time usuário ou usuário final. Para o primeiro caso, poderia ser "Como podemos permitir que o time XPTO forneça mais tempo para os usuários finais interagirem com a funcionalidade X?"; para o segundo, "Como podemos permitir a interação da funcionalidade X pelos usuários finais?".

Ainda há a opção de enquadrar de forma mais abrangente, já que plataformas precisam atender a contextos diferentes. Uma ideia poderia ser "Como podemos tornar o limite de tempo variável de acordo com cada tipo de operação?". Também resolvemos problemas de engenharia, então os participantes da sessão podem trazer questões técnicas, por exemplo, "Como podemos evitar que o microsserviço X fique indisponível?".

14.2 Brainstorm

Embora muitos de nós já tenhamos participado de *Brainstorm*, nem sempre é fácil facilitar uma sessão que seja realmente produtiva. Colocar uma tela ou folha de papel em branco e pedir que o time apresente ideias não costuma funcionar bem, assim como depender apenas da discussão. Uma dinâmica visual funciona melhor. Escrever ou desenhar é uma forma de tangibilizar a ideia e pode ajudar as pessoas a verem-na de uma perspectiva diferente ou a terem novas ideias.

Você ou alguém do time pode escrever ou desenhar as ideias dos participantes no quadro branco, parede ou *whiteboard*. Pode ser em qualquer formato: texto, bullets, diagrama, esboço, *print* de alguma referência etc. Se o processo desacelerar, você, como pessoa

facilitadora da dinâmica, pode fazer perguntas para estimular a discussão, como "e se um dos times que vão usar a plataforma não souberem como integrar à API assíncrona?" ou "como poderíamos tornar a integração mais *self-service* para devs juniores?".

Depois disso, as ideias são refinadas e os participantes selecionam as que parecem as melhores soluções, afunilando-as nas mais práticas, factíveis, viáveis ou mais inovadoras dentre as opções apresentadas.

É importante que seja restrito a um problema por vez e é uma boa definir um tempo. Do contrário, a sessão perde o foco, as discussões ficam longas e se tornam improdutivas.

Caso baseado em uma história real: Card sorting

Havia aproximadamente 3 meses que eu estava no time da Order Platform quando fui chamada para uma conversa com alguns dos nossos usuários. É uma plataforma tão difícil, tão complexa e eram tantos assuntos diferentes acerca da versão vigente e da nova versão que nesse período de tempo aprendi pouco, comparado com a totalidade. Acredito que eu entendia apenas um terço dos assuntos que rolavam em três squads (totalizando 18 pessoas e eu, a única PM).

Nessa conversa, um dos engenheiros apresentou a proposta de solução para o *payload* — dados que trafegam entre os sistemas — da nova plataforma. Ele mostrou a documentação, as mudanças, e os times usuários comentaram, mas saímos dela sem nada concreto e com insegurança sobre se poderíamos seguir com o que foi proposto.

Foi então que me lembrei do *Card sorting* — técnica tradicionalmente usada para definir a arquitetura da informação de produtos *customer-facing* —, e pensei: por que não o usar para plataformas? Nesse método, atribui-se cada ideia de conteúdo a

cartões de índice e os distribui de forma aleatória, como na próxima figura.

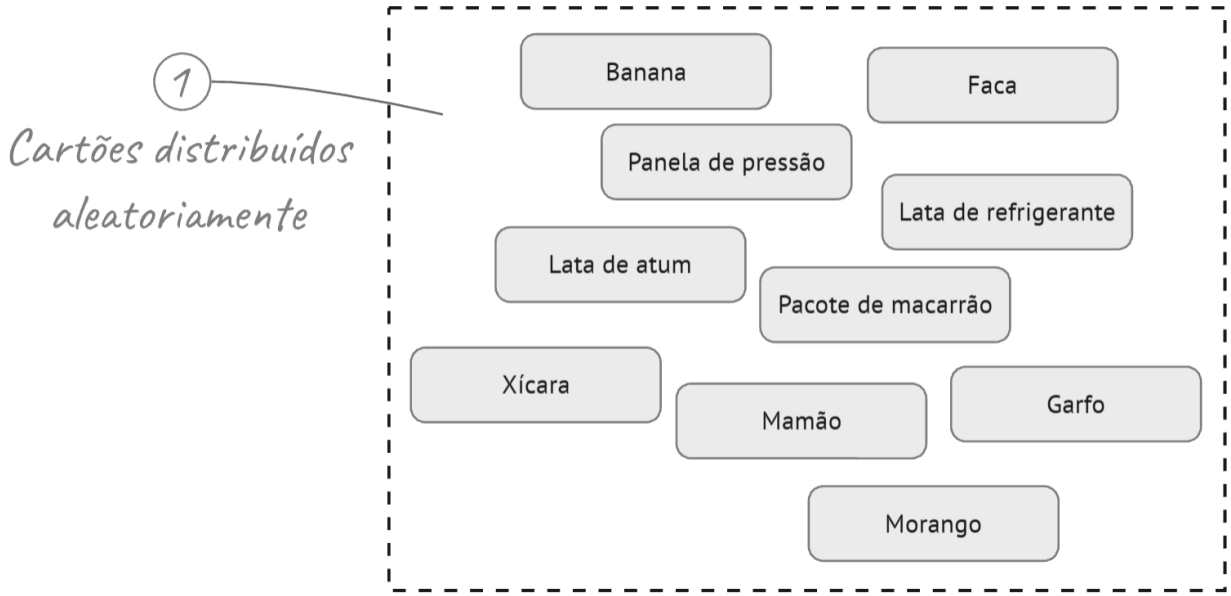


Figura 14.1: Cartões de índice distribuídos aleatoriamente.

As pessoas agrupam os cartões por algum tipo de semelhança que faça sentido para elas e depois nomeiam os grupos, como exemplifica a figura adiante.

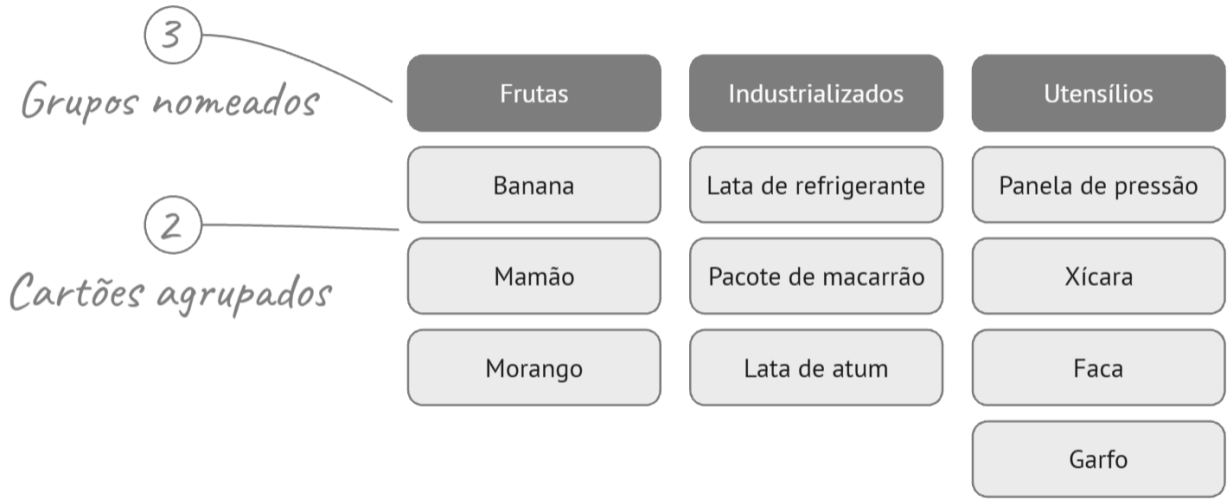


Figura 14.2: Cartões agrupados e grupos nomeados.

Em linhas gerais, o problema da Order Platform era que a estrutura do *payload* era composta por atributos em excesso. Também havia duplicação de nomenclatura e isso deixava o significado confusos. Os dados são gerados por diferentes apps e plataformas, e os times do iFood não sabiam quem era responsável por cada tipo de dado. Faltava documentação, o que demandava muito suporte do time da Order Platform.

Veja que não era a criação de algo do zero. Existia o *payload* em produção e era utilizado por todos os times usuários. Além disso, já havia uma proposta para a nova plataforma que não tinha motivo para ser ignorada. Então, fizemos sessões de *card sorting* em cocriação com os times usuários. Organizei um *board* no Miro com a proposta do novo *payload* — aquela que foi apresentada a alguns times clientes. Para isso, usei a ferramenta *user story map*, mas não para visualizar uma jornada; dessa vez, essa ferramenta foi escolhida pela facilidade de movimentar e encaixar os cards.

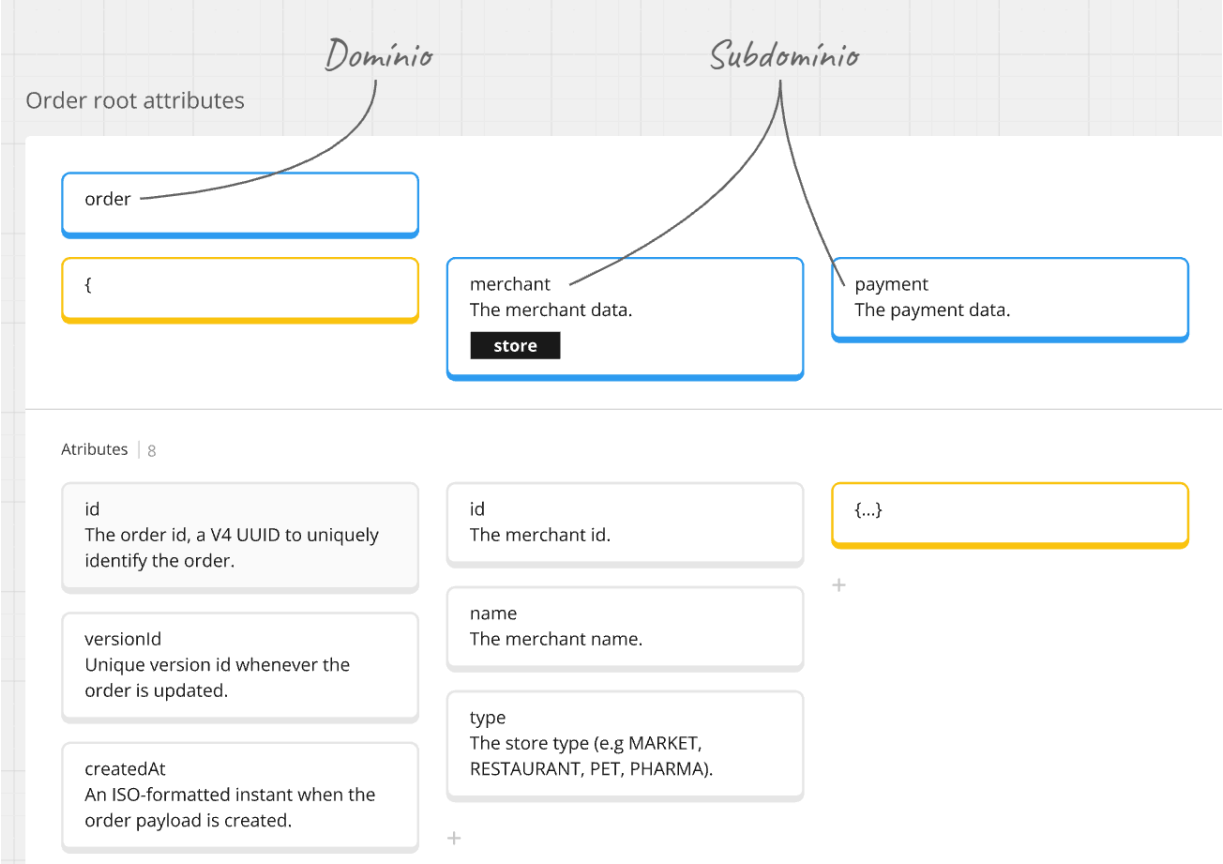


Figura 14.3: Preparo para a dinâmica de Card Sorting.

O card "*order*" refere-se ao domínio que contempla todos os dados atrelados a um pedido. A chave do card abaixo dele significa que o que virá na vertical e horizontal são os dados na raiz. Os da horizontal são subdomínios. O card do subdomínio "*merchant*" possui a tag "*store*". Isso indica que foi renomeado de *store* para *merchant*. O objetivo dessa mudança era deixar a nomenclatura genérica, dessa forma ficaria coerente com dados de pedidos qualquer segmento. Abaixo de "*merchant*" vemos cards com os respectivos atributos na vertical. O card com o símbolo `{...}`, representa que o subdomínio "*payment*" possui mais níveis de atributos encadeados. Complicou? Imagine que, se o payload fosse um texto comum, esse exemplo teria a seguinte estrutura hierárquica:

```
- order
    - id
    - versionId
    - createdAt
    - merchant
        - id
        - name
        - type
    - payment
        - totalValue
        - invoiceId
        - methods
            - amount
            - type
            - card
                - brand
                - number
                - tokenId
            - ...
```

Como a ferramenta de *user story map* do Miro não habilita a inserção de mais camadas no nível superior, os subdomínios com mais níveis foram separados. Depois de tudo organizado, chamamos os times para colaborar, o que resultou em algo como a figura a seguir:

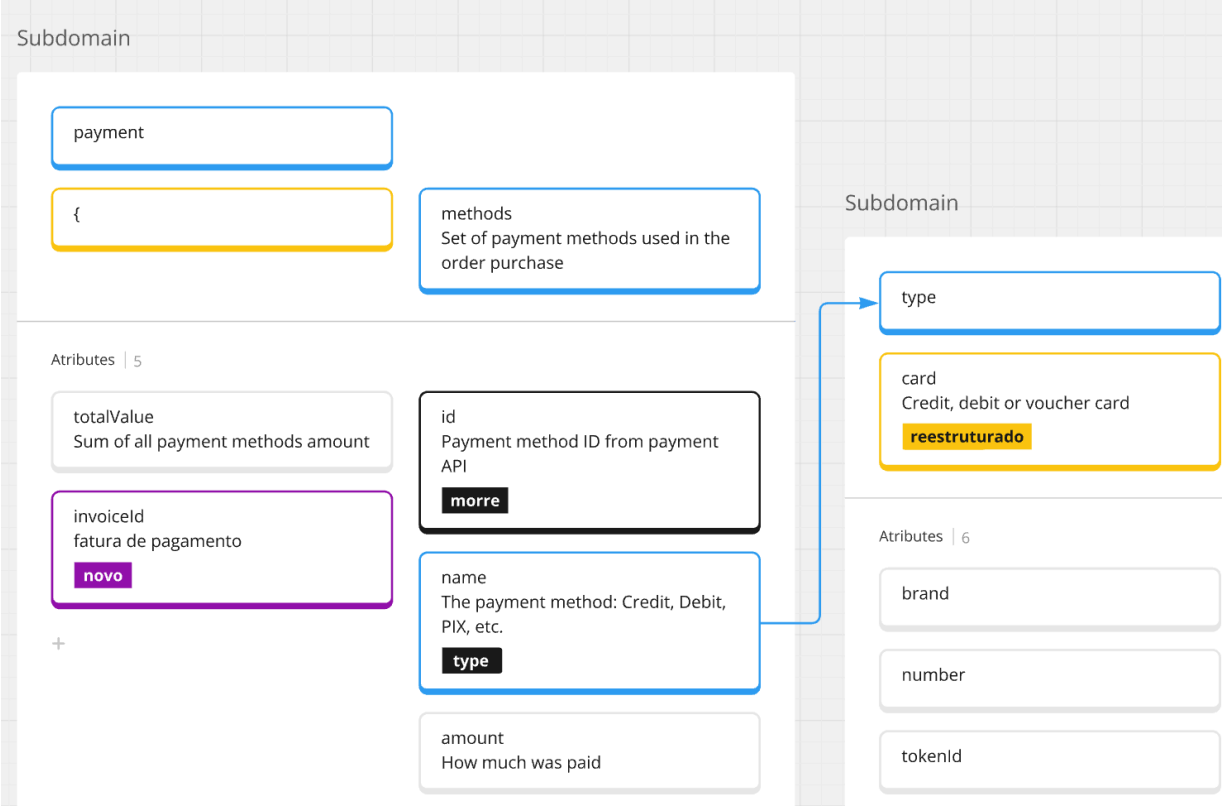


Figura 14.4: Cocriação com os times usuários

A tag “morre” significa que o atributo deveria ser retirado do *payload*; já a tag “novo” significa que o atributo não existia no payload em produção, nem na proposta para a nova plataforma, portanto seria adicionado.

Foram necessárias várias sessões com 17 times usuários — vez ou outra participava mais de um time. Nessas sessões, 60 atributos que não eram utilizados foram eliminados e vários foram renomeados e/ou reestruturados.

Durante a organização do *board*, percebi alguns atributos duplicados, o que me fez pensar que a proposta de solução não estava bem resolvida. Imagine que os métodos de pagamento eram como os da figura a seguir. A duplicação para cada tipo de cartão deixava o *payload* maior do que o necessário — e isso acontecia para vários outros atributos.

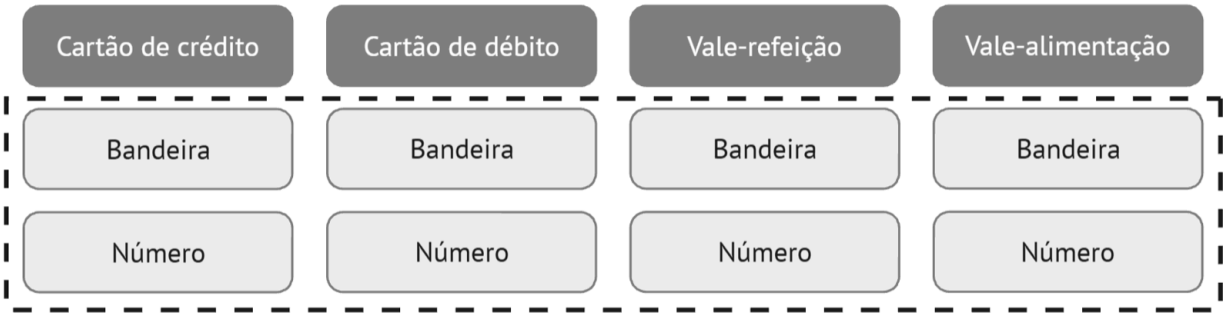


Figura 14.5: Atributos duplicados.

Na sessão com o time responsável pela plataforma de pagamentos, questionei se isso não poderia ser unificado, e chegamos a algo como a próxima imagem.

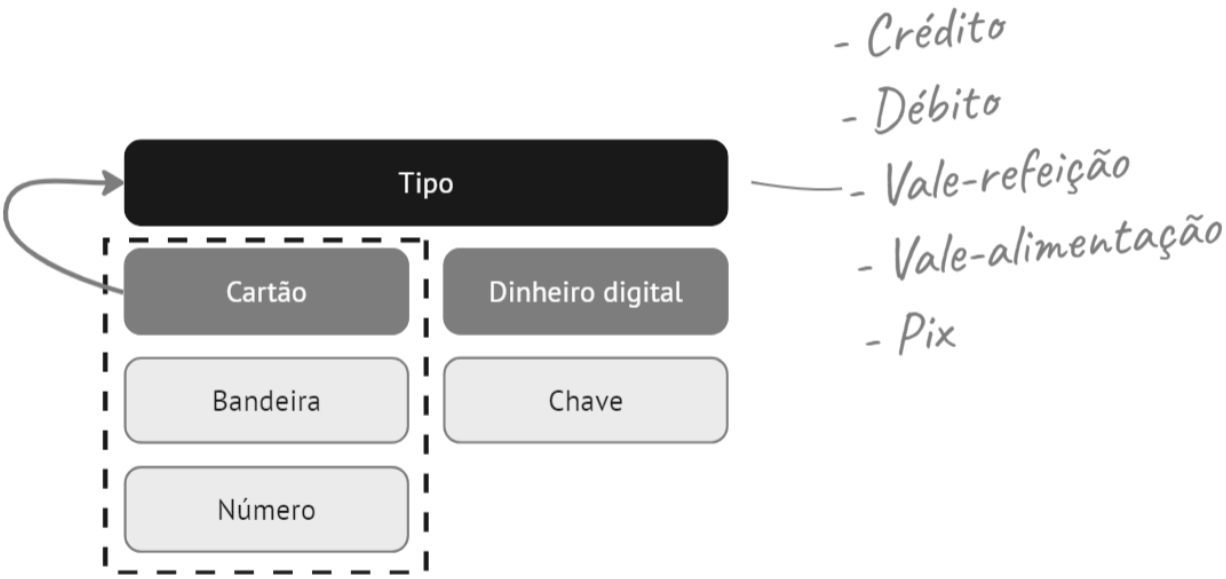
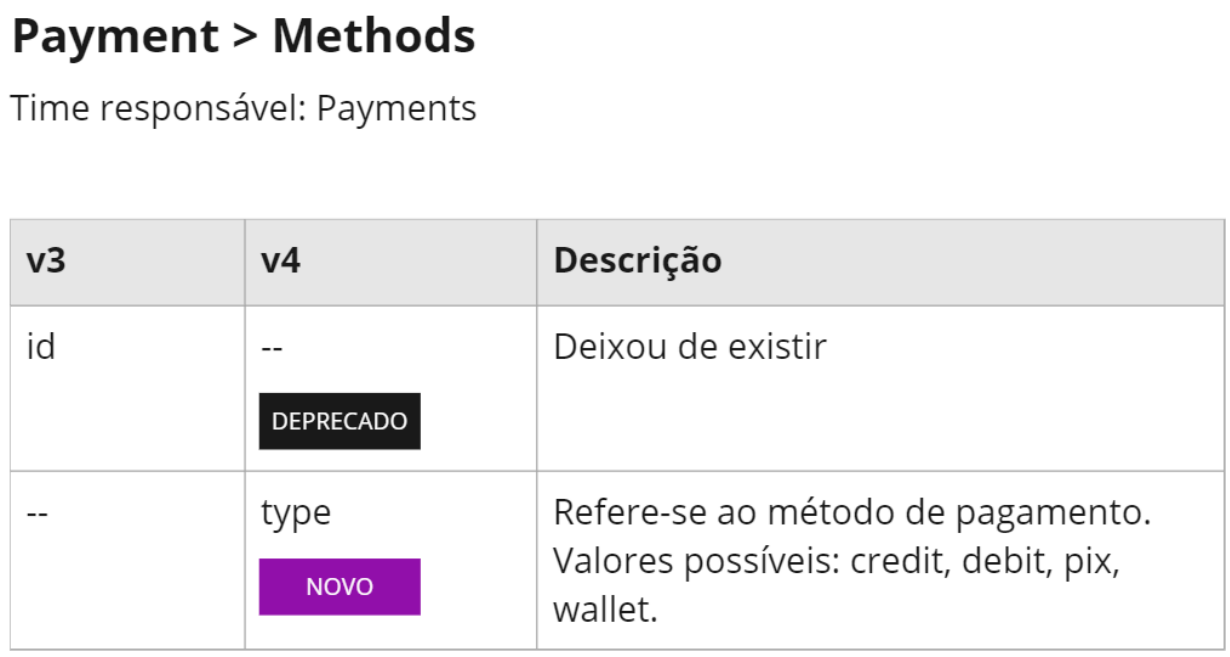


Figura 14.6: Esquema reestruturado.

As duplicações foram sintetizadas em "Cartão" e, com essa mudança, foi adicionado o atributo "Tipo" na hierarquia, que recebe determinado valor para distinguir o método de pagamento. Se o valor for "Crédito", por exemplo, atrela ao atributo "Cartão" e os respectivos dados; se for "Pix", atrela ao "Dinheiro digital". A escolha pelo nome "Dinheiro digital" foi a tentativa de deixá-lo genérico, afinal, não sabemos o que o futuro reserva, não é mesmo? Daqui a pouco aparece algum novo método e seríamos obrigados a mudar o

payload. Esse tipo de mudança exige alterações em todas as aplicações integradas à plataforma. Já o valor — débito, crédito, pix etc. — pode ser adicionado infinitamente, sem mudanças na Order Platform ou qualquer outro sistema.

Além da implementação do que foi definido nessas sessões com os times, precisávamos documentar, porque seria usado para plugar na plataforma e também para análise dos dados. A imagem a seguir dá uma noção de como organizei a documentação e as diferenças entre os dados da plataforma vigente e a nova. Repare que há o time responsável pelo domínio dos dados para que as pessoas saibam qual time procurar em caso de dúvida ou problemas.



Payment > Methods

Time responsável: Payments

v3	v4	Descrição
id	-- DEPRECADO	Deixou de existir
--	type NOVO	Refere-se ao método de pagamento. Valores possíveis: credit, debit, pix, wallet.

Figura 14.7: Documentação para uso dos times.

14.3 Árvore de Oportunidades

A Árvore de Oportunidades — *Opportunity Solution Tree*, proposta pela Teresa Torres (2016) — ajuda a estruturar as ideias que surgem no *brainstorm*. A figura adiante exibe como ela sugere que a árvore

seja estruturada. No topo, o **resultado desejado** (*desired outcome*), que pode ser o resultado-chave (*key result*), quando se usa OKR, ou qualquer métrica que se queira melhorar. Ligadas a ele, as **oportunidades** (*opportunity*), em outras palavras, os problemas que possuem o potencial de trazer valor. Em seguida, as ideias de **solução** (*solution*) para cada oportunidade. Por fim, os **experimentos** (*experiment*) para testar as hipóteses de solução.

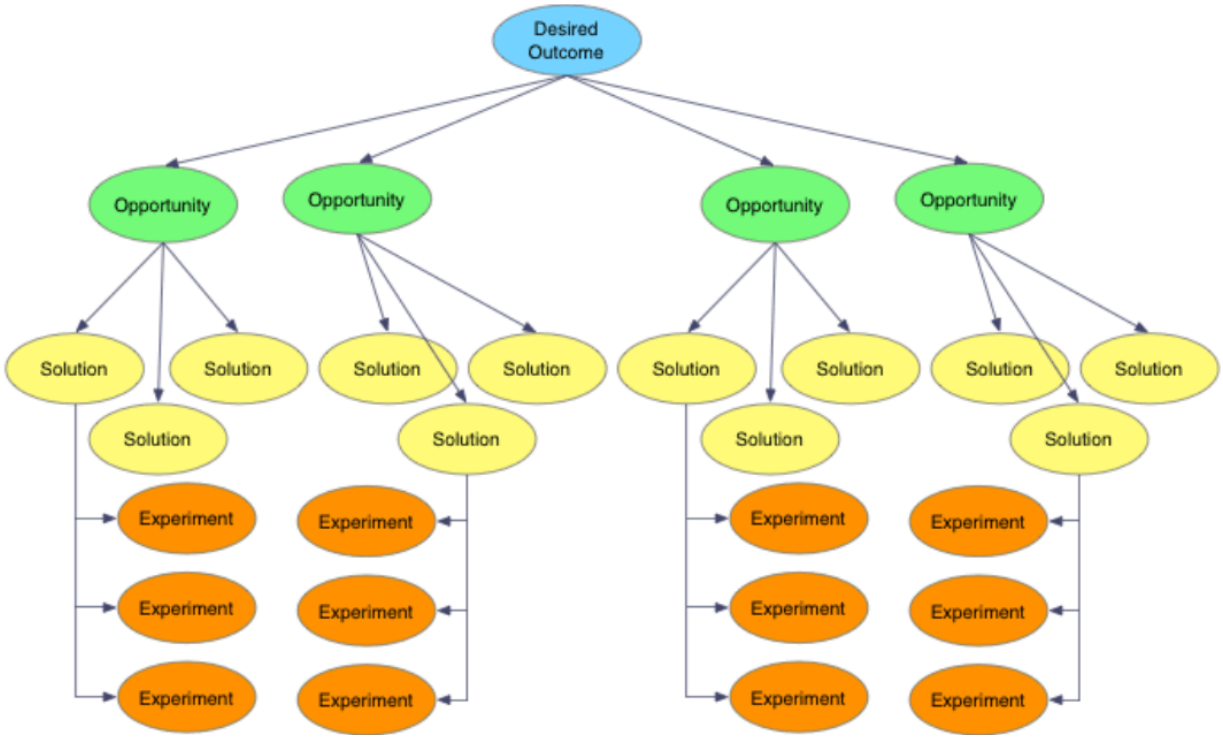
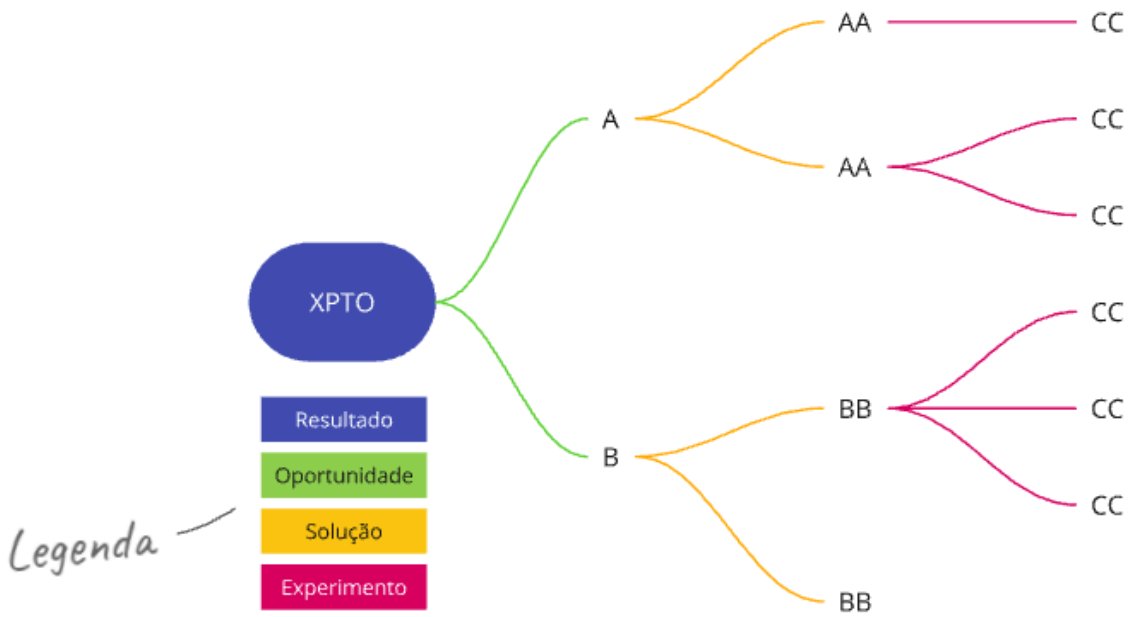


Figura 14.8: Opportunity Solution Tree. Fonte: TORRES, 2016.

Perceba que a Árvore de Oportunidades nada mais é do que um mapa mental, então, é a ferramenta que eu uso no Miro, e diferencio as camadas — resultado desejado, oportunidades, soluções e experimentos — por cores com legendas ou por delimitação de área com título, como nas próximas imagens.


XPTO
A
B
AA
AA
BB
BB
CC
CC
CC
CC
CC
CC
Resultado
Oportunidade
Solução
Experimento
Legenda

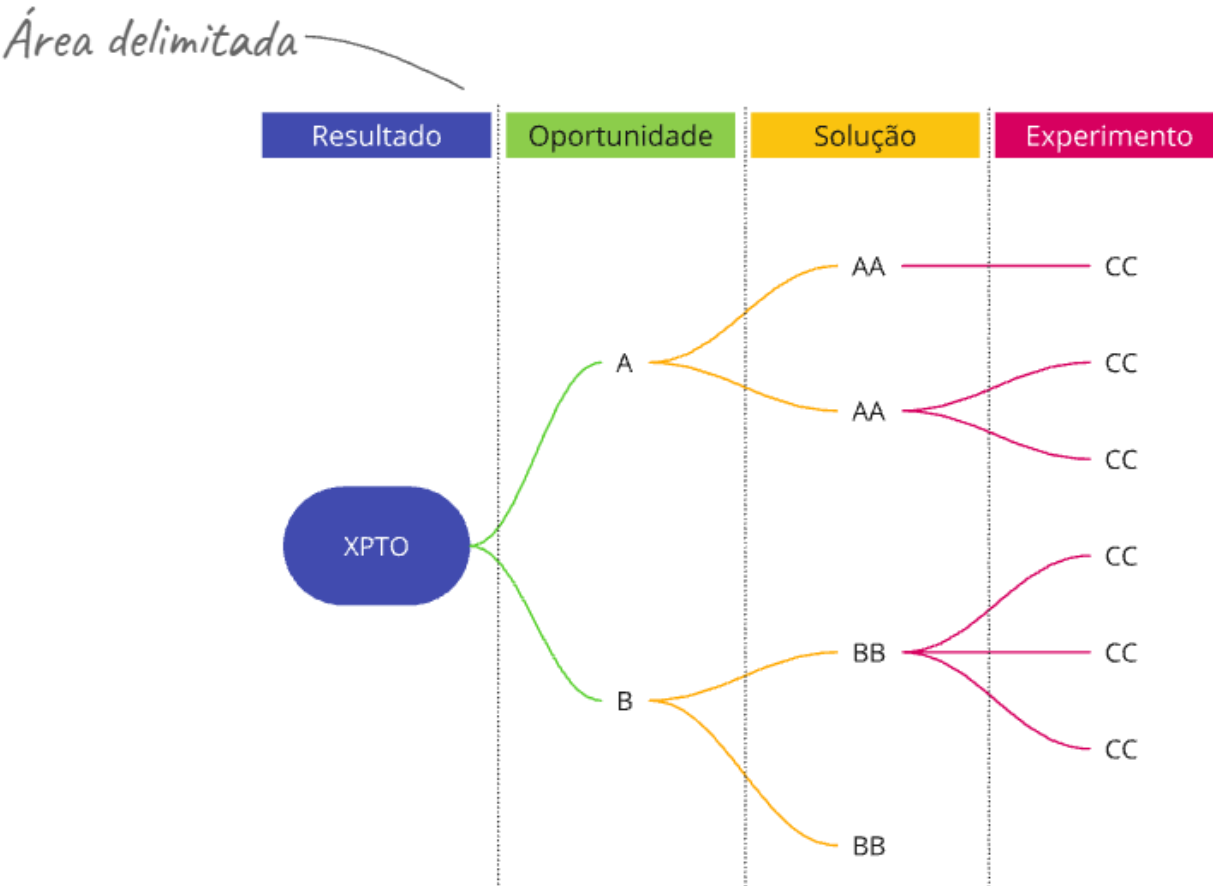

Área delimitada
Resultado
Oportunidade
Solução
Experimento
XPTO
A
B
AA
AA
BB
BB
CC
CC
CC
CC
CC
CC

Figura 14.9: Árvore de Oportunidades.

Qualquer ferramenta ou framework ajuda a estruturar as coisas, e às vezes é preciso adaptá-los para nos servir. Neste caso, podemos substituir o resultado desejado e/ou oportunidades pela declaração do problema ou a questão "Como podemos…?".

No exemplo da imagem adiante, a *oportunidade* foi substituída pelo *Como podemos…?* e adicionada uma restrição no *post-it*. Vamos supor que uma das soluções esbarre na restrição. Em uma situação como essa, deixo visual que a ideia não seguirá no processo e um comentário do porquê.

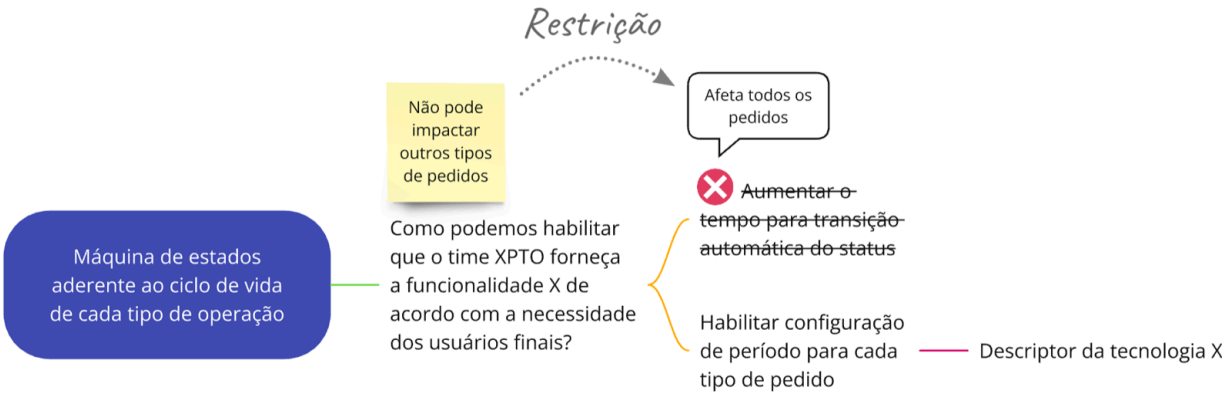


Figura 14.10: Árvore de Oportunidades adaptada.

Para levantamento das ideias de solução, podemos trazer *benchmark* de plataformas e APIs que resolvem problemas semelhantes, mesmo que de outro setor. É tão simples quanto entrar nos sites das empresas, ver a proposta de valor, as documentações das APIs ou os blogs de engenharia onde compartilham os aprendizados, por exemplo:

- Uber → https://www.uber.com/en-US/blog/engineering/
- Netflix → https://netflixtechblog.com/
- Shopify → https://shopify.engineering/

Se nessa etapa chegou-se a várias ideias, selecione aquelas que parecem mais promissoras. Para isso, você pode usar métodos que

você já conhece, como *Dot voting* ou algum critério, como *Matriz esforço x impacto*. Feito isso, entramos no próximo estágio do Design Thinking, a *Prototipagem* da solução.

CAPÍTULO 15
Prototipagem: Criação de soluções

Uma das melhores maneiras de entender como os usuários vão interagir com um produto é colocar o produto nas mãos deles para que possam testá-lo. Desenvolver um produto acabado somente para testes é caro e demorado e muitas vezes investe-se em ideias e, após lançá-las, não se obtém tração de usuários. Isso acontece porque as suposições que basearam as soluções podem estar erradas. Quando erradas, levam a perdas significativas de tempo e recursos.

A *Prototipagem* ajuda a evitar isso. Ela serve para testar as suposições, aprender sobre os usuários e aprimorar as soluções. É o que dá vida às ideias, transforma coisas abstratas em algo tangível, mais próximo da realidade, o que permite endereçar com mais rapidez os riscos de usabilidade e viabilidade técnica. É essencial no design da experiência, e os feedbacks dos usuários na interação com o protótipo na etapa de *Teste* servem de evidência para atestar se vale a pena construir a solução e se será possível entregá-la aos usuários.

Prototipar envolve a produção (iterativa) de versões experimentais, rápidas, baratas e simples do produto, com objetivo de identificar a melhor solução possível — dada as restrições —, em que pouco tempo e dinheiro sejam investidos em uma ideia que tem a possibilidade de se revelar ineficaz.

Um protótipo pode ser qualquer coisa que concretize as ideias e torne-as testáveis. Isto é, que o usuário consiga interagir e experimentar — rascunhos em papel, desenhos esquemáticos, *wireframes*. Pode-se prototipar uma parte específica da solução ou recurso; não precisam ser soluções completas.

Os métodos de prototipagem são geralmente divididos em duas categorias: protótipo de **baixa fidelidade** e **alta fidelidade**, em que *fidelidade* se refere ao nível de detalhe e interatividade. Há quem considera a existência de um nível intermediário, *média fidelidade*.

Nas fases iniciais, é comum a produção de protótipos de baixa resolução, mais rápidos e baratos de se fazer (coisa de minutos), mas suficientes para obter *feedback* útil. Tanto o protótipo quanto as questões a serem respondidas por meio dele vão se refinando conforme essa etapa avança, podendo chegar ao protótipo de alta fidelidade.

Espera-se que ninguém duvide da necessidade de testes de usabilidade quando o assunto é interface visual. Temos a capacidade de avaliar a usabilidade de qualquer produto, seja ele físico ou digital. Por que não de plataformas e APIs? Com certas adaptações, é possível usar os conceitos dos métodos já conhecidos para produtos *customer-facing*.

15.1 Protótipo de baixa fidelidade

Na prototipagem de baixa fidelidade, são produzidas versões simples, muitas vezes com materiais disponíveis no momento — papel, *post-it*, caneta, adesivos —, e que podem ser testadas imediatamente. Pode estar incompleto e exibir apenas alguns dos recursos que estarão disponíveis no produto final. Além de barato e fácil de fazer, deve ser passível de alterações instantâneas e de testes das iterações. Para isso, eles são produzidos com poucos detalhes, são descartáveis (não dá pena de jogar fora), o que permite que se obtenha uma visão geral da solução usando o

mínimo de tempo e esforço, em vez de dar atenção a detalhes que tomam tempo para alterar.

Esse tipo de protótipo tira o controle do usuário, pois permite apenas interações básicas, simuladas e o usuário apenas informa como usaria o produto acabado. Por isso, é mais comumente usado antes do desenho do protótipo de alta fidelidade.

APIs podem ter características experimentais, mas dificilmente é possível fazer experimentos no *core* da plataforma sem impactar gravemente o ecossistema. Mas tem uma boa notícia: pode ser tão simples quanto o protótipo no papel.

O caso baseado em uma história real do *card sorting*, visto no capítulo anterior, pode ser considerado um protótipo de baixa fidelidade. Você pode questionar que não houve teste de usabilidade. Nesta situação, o design do *payload* foi cocriado com os times que já usam a solução, portanto, eles conseguem tangibilizar o resultado final durante o processo.

Agora vejamos dois outros casos baseados em história real. Um deles foi apresentado à alta gestão; o outro gerou um protótipo de baixa fidelidade que foi levado para teste com os usuários da Order Platform.

Caso baseado em história real: Wireframe de arquitetura

Wireframes são ilustrações simples, com elementos básicos da solução, isto é, ignoram os aspectos visuais e interativos, como a figura a seguir — você já deve estar familiarizado(a). O foco é a funcionalidade e a estrutura do conteúdo, tornando as alterações rápidas e fáceis, em comparação com protótipos de alta fidelidade. Esse formato mantém os usuários atentos aos aspectos funcionais, em vez de preferências visuais.

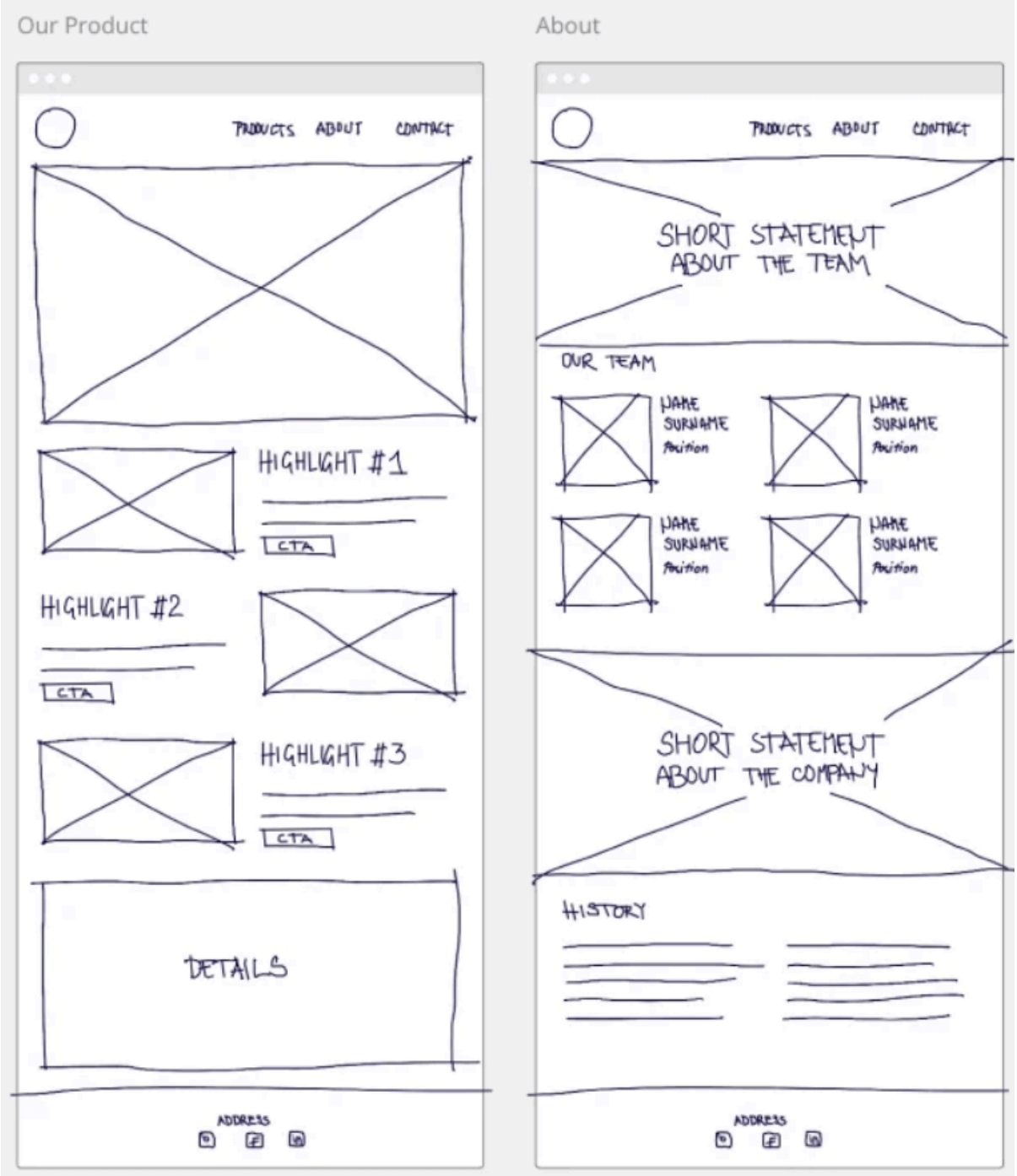


Figura 15.1: Exemplo de wireframe de baixa fidelidade. Fonte: Low-fidelity Wireframes Template (MIRO, c2023).

Como você imagina que seria um *wireframe* para código de programação? Antes de responder, é importante dizer que você não vai encontrar por aí o termo "*wireframe*" para plataformas ou APIs.

Passei a chamar dessa maneira os que veremos agora com base no conceito retratado anteriormente. Isso ajudou na comunicação com o time da Order Platform e talvez a referência faça sentido para você também. Então, para mim, se código é texto, o *wireframe* só pode ser texto também! Você saberá em detalhes o *wireframe* de uma API no capítulo *Da ideação ao teste de usabilidade*.

Quanto à arquitetura do software, chamo de *wireframe* o que o time da Order Platform utiliza para elaborar o fluxo, interação entre os sistemas e tecnologias — algo parecido com os diagramas a seguir:

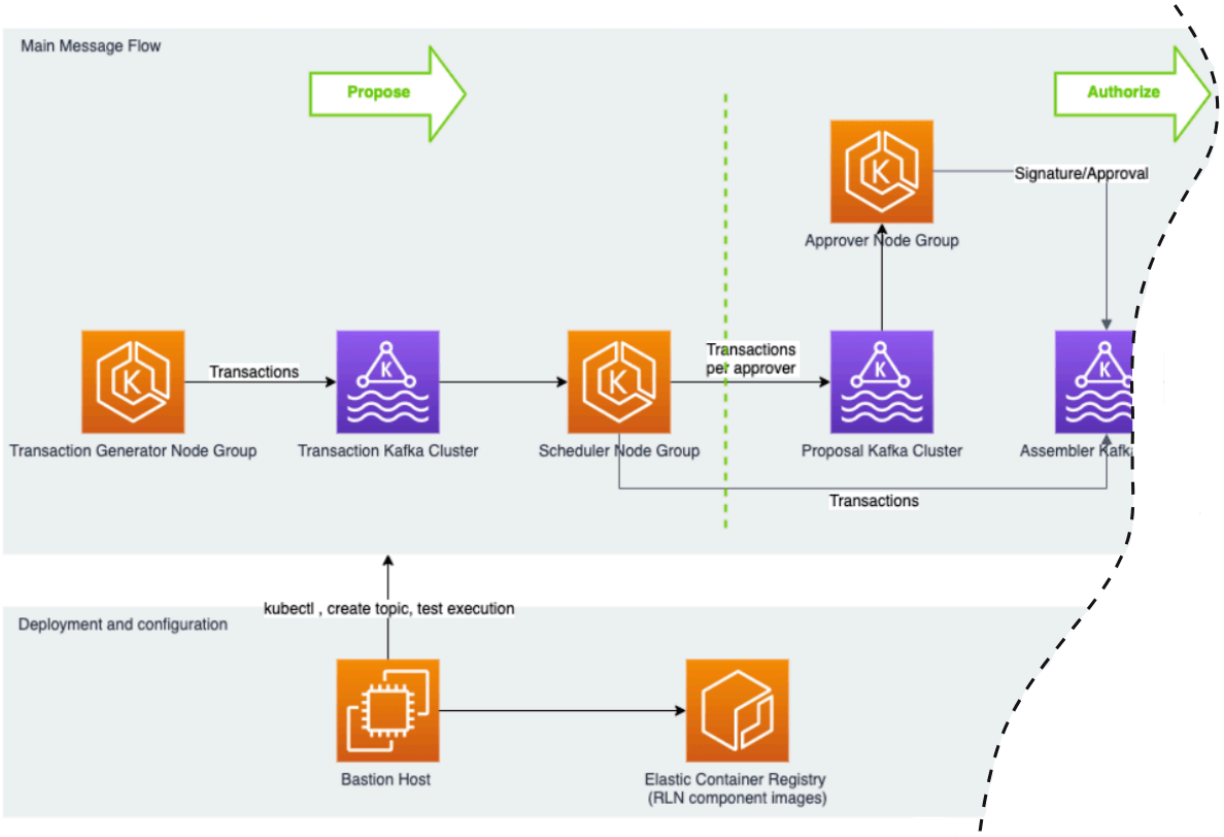


Figura 15.2: Exemplo de diagrama de arquitetura. Fonte: AWS Architecture Blog (SALINAS; LU, 2022).

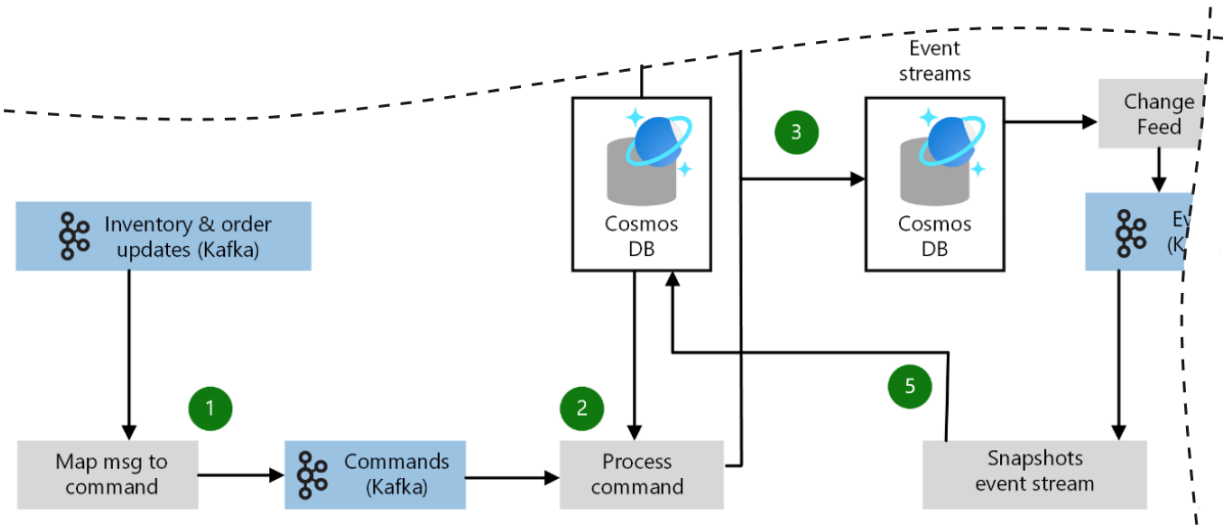


Figura 15.3: Exemplo de diagrama de arquitetura. Fonte: Azure Architecture Center (c2023).

Normalmente, as pessoas de engenharia estão acostumadas com diagramas e os devs usuários conhecem bem os casos de uso e o código da aplicação deles, então, com a devida explicação dos elementos do "*wireframe*", eles conseguem interpretar e imaginar o resultado com facilidade. Também é útil para comunicação com as pessoas da alta gestão de engenharia. Elas compreendem claramente o que vai acontecer, onde, como e conseguem contribuir.

No caso da Order Platform, alguém de engenharia de alta senioridade cria o diagrama e depois o apresenta para receber feedback do time, semelhante ao *design critique* de produtos *customer-facing*.

Pode ser difícil para PMs entenderem isso quando começam em gestão de plataforma. Isso aconteceu comigo e aproveitei a oportunidade do "*design critique*" para aprender sobre as tecnologias à medida que o time discutia. Quando eu percebia que mencionavam restrições, eu apenas perguntava se ia atender a determinada necessidade ou caso de uso. A depender da situação, eu pedia para alguém do time me explicar depois, para não prejudicar a produtividade da sessão.

Caso baseado em história real: Wireframe de feature

No capítulo *Empatia*, vimos o caso da jornada dos usuários finais. Os desenhos das jornadas de diferentes tipos de pedidos foram a base para definir os status da máquina de estados da nova plataforma.

Para o workshop com o time da Order Platform, preparei um *board* no Miro com os status da plataforma atual. A próxima imagem é a amostra de um pequeno exemplo:

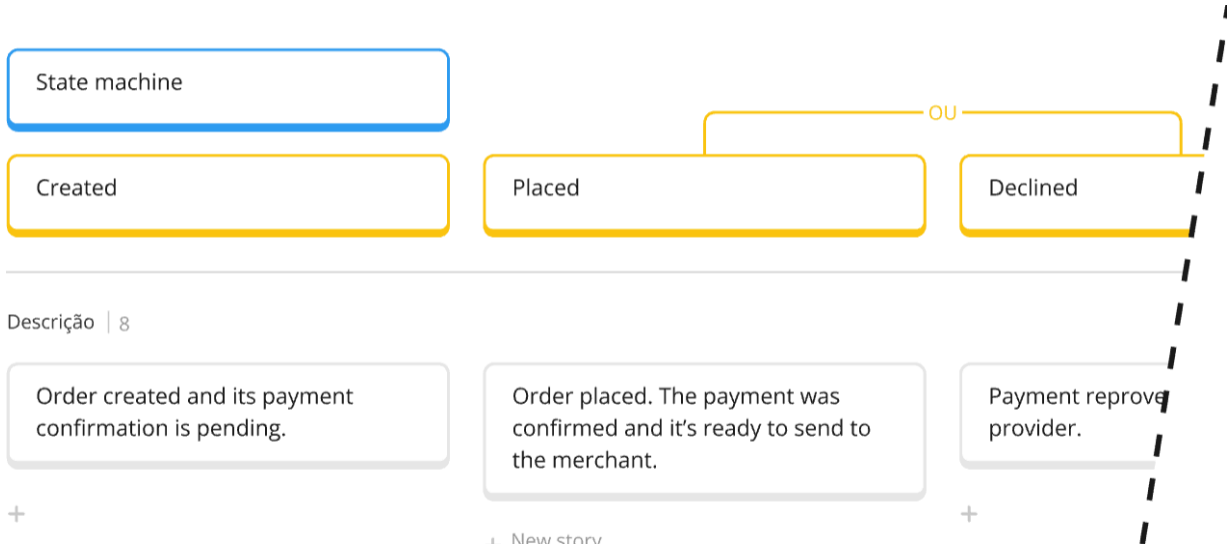


Figura 15.4: Preparo para o workshop.

A primeira coisa durante a sessão foi compartilhar as premissas e expectativas dos stakeholders acerca da solução. O time ajudou a completar com aspectos técnicos e o resultado é uma lista em forma de texto.

PREMISSAS

- Status com significados únicos em todo o iFood;
- Status inicial e final únicos;
- Não volta status (para o passado);
- Não altera o tipo de pedido e/ou versão depois de criado;
- Flexível, configurável e escalável;
- Genérico para processar pedidos além de refeição.

Isto feito, passamos por cada status para avaliar se a nomenclatura era coerente com o significado. Considerando as necessidades dos times, alteramos nomes, atribuímos o devido significado, adicionamos casos de uso e regras, além da criação de novos. Para nos apoiar na ideia de solução, separei alguns *benchmarks*, como o diagrama adiante do Google Shopping, e, como facilitadora ao longo da sessão, eu voltava a ele e a outras referências para discutirmos.

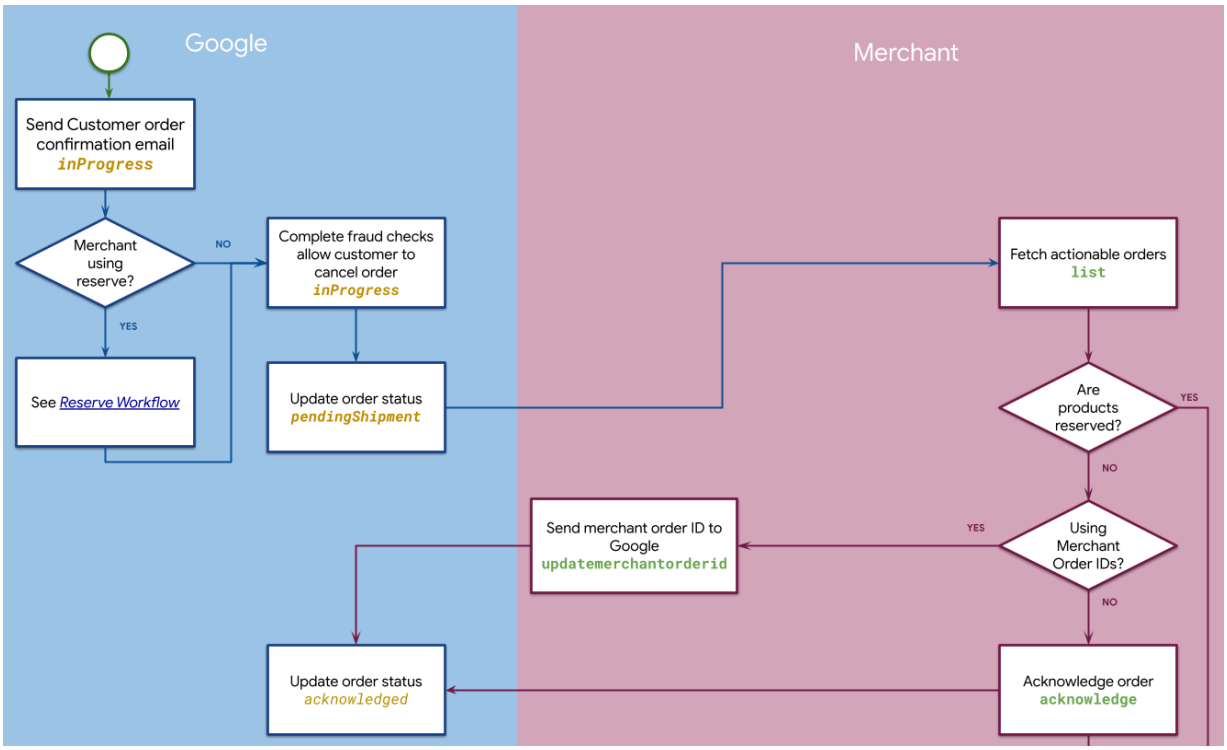


Figura 15.5: Processo do pedido do Google Shopping. Fonte: Content API for Shopping (c2023).

Depois de algumas sessões com o time, emergiram dúvidas de como os novos status e novas formas de uso iriam afetar outras plataformas. Estava previsto eliminar alguns deles, e a grande descoberta foi a de que um status específico não poderia ser eliminado, porque seria necessária uma adaptação drástica em algumas plataformas.

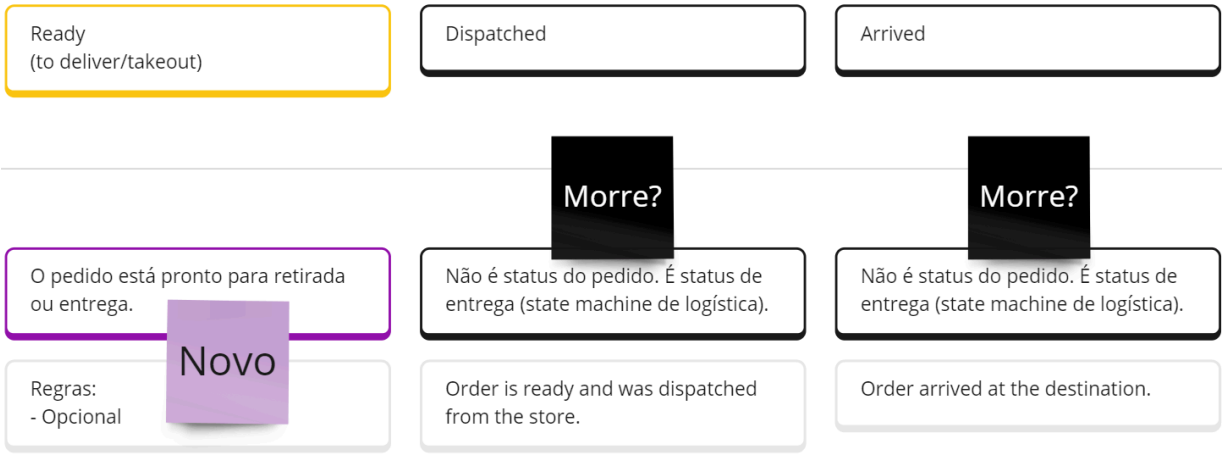


Figura 15.6: Proposta de solução.

Uma vez validado com os times das plataformas, compartilhamos a versão final com os times dos apps. Para isso, foi criado o que chamei de "protótipo", que ajuda a tangibilizar como ficaria a máquina de estados para cada tipo de pedido em que cada time atuava. Apresentamos aos times para coletar *feedback*, e claro que chamei de "teste de usabilidade". Esse modo de apresentação não traz grandes garantias como um verdadeiro teste de usabilidade. No entanto, para tornar o protótipo funcional, neste caso, precisaria desenvolver de fato, o que levaria tempo demais. Este formato foi suficiente para obter um parecer se seria útil para os cenários conhecidos, pelo menos. A próxima figura é o exemplo de um trecho do "protótipo" de uma das unidades de negócio que foi apresentado ao respectivo time.

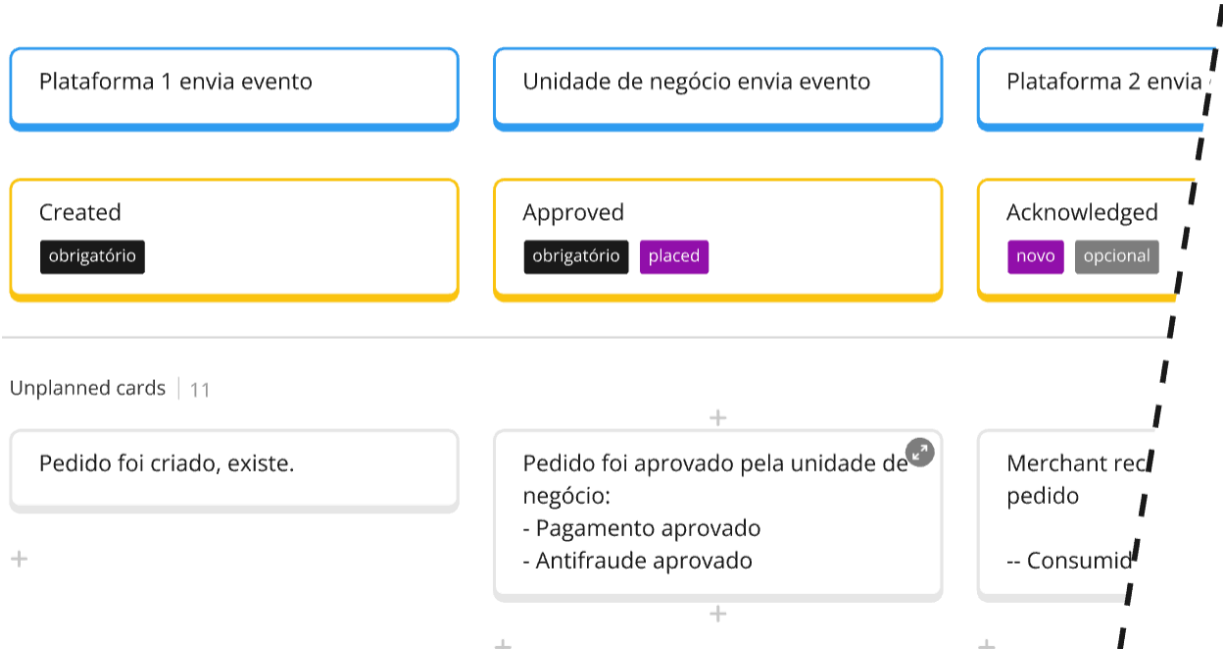


Figura 15.7: Proposta de máquina de estados para uma das unidades de negócio.

Acabaram de ser apresentados dois casos em que as ideias se tornaram mais tangíveis nas conversas com os devs usuários. Em ambos, apesar de os protótipos serem estáticos e com certo nível de incerteza — parece improvável que haja uma maneira de criar protótipos interativos para esses cenários — são úteis, pelo menos, para validação de valor. A certeza transparece somente depois que se disponibiliza a solução para uso.

15.2 Protótipo de alta fidelidade

Protótipos de alta fidelidade costumam ser mais próximos da solução final em termos de aparência e sensação (*look and feel*), pois permitem que os usuários interajam da mesma maneira que no design real. Assim, stakeholders conseguem avaliar o quão bem a solução atende às suas expectativas, desejos e necessidades.

Quanto mais semelhante for o protótipo em relação à solução acabada, mais confiança se tem na validação de como as pessoas

vão responder, interagir e perceber o design da solução.

Quando falamos de protótipo de produtos *customer-facing*, é esperado um fluxo com design interativo, para proporcionar ao usuário a mesma sensação de uso do produto. E quando o assunto é plataforma?

Protótipo de API

É possível prototipar APIs e deixá-las interativas, sem desenvolver o back-end — uma baita economia de tempo e esforço. Existem ferramentas que geram a documentação da API para compartilhar com os devs usuários, o protótipo interativo para enviar uma requisição (*requests*) e receber a resposta (*response*) em tempo real. O capítulo *Da ideação ao teste de usabilidade* traz tudo sobre isso.

Protótipo de back-end

A PoC (*Proof of Concept*, em português, prova de conceito) nada mais é do que um experimento contido — um pequeno desenvolvimento (código) com testes para validar determinado requisito. Neste caso, não é testado pelos usuários, é para diminuir incertezas acerca da viabilidade técnica.

Se a PoC demonstrar que a solução é tecnicamente viável, guiará o "como" e trará uma noção de tempo para implementação na fase de *delivery*. Claro que pode demonstrar que é inviável, então, passa-se para a validação de outras alternativas de solução. Portanto, considerando que PoCs validam ideias de modo mais próxima da realidade, você concorda que podemos considerar um protótipo de alta fidelidade? Por isso, vale a pena vincular como uma atividade de *discovery*. Executar PoCs no *delivery* pode ser tarde demais para descobrir que a solução é inviável.

Independentemente do tipo de protótipo que você e o time optarem por produzir, é importante definir com antecedência o que se

pretende testar com o usuário, que tipo de comportamento é esperado, o que precisa de resposta que leve à decisão de seguir adiante, abandonar ou melhorar a solução. Essas definições ajudam a manter o foco da *Prototipagem* no que necessita ser descoberto e receber feedback relevante na fase de *Teste*, a etapa do próximo capítulo.

Capítulo 16
Teste: Validação das hipóteses

Protótipo pronto, entramos na quinta e (teoricamente) última etapa do Design Thinking, o *Teste* com os usuários para coletar *feedback*. Os *feedbacks* respondem se a solução proposta entrega valor e se, na perspectiva deles, é fácil de usar. Essa etapa é também uma oportunidade para exercer a *Empatia* e conhecer os usuários ainda melhor. Mais aprendizado sobre eles e sobre os problemas podem trazer *insights* inesperados para possíveis soluções.

Antes do teste, é recomendado se preparar para manter a descoberta centrada no que você precisa de informação para decidir o próximo passo: desenvolver a solução, abandoná-la ou melhorá-la.

16.1 Planejamento do teste

Não podemos simplesmente colocar um protótipo na frente do usuário e ver o que acontece. Precisa haver direcionamento sobre o que precisa ser testado.

As etapas de *Prototipagem* e *Teste* são interligadas. É comum se ter noção do que tem que ser testado — justamente por isso é contemplado no protótipo — e, depois do protótipo pronto, pode-se perceber outras coisas importantes a se testar. Quando isso acontece, é o momento de refinar o que foi planejado previamente. Nos casos em que não houve planejamento, é hora de definir algumas coisas.

Critérios para avaliação

Como o *feedback* será interpretado? Para responder a essa pergunta, é preciso especificar critérios para avaliação. Os critérios

dependem do tipo de protótipo. Para APIs, é provável que se queira saber se é fácil de usar e se a documentação está clara. Já PoCs validam se a implementação é viável. Uma é totalmente relacionada à usabilidade e experiência do dev, enquanto a outra é pura engenharia e envolve máquinas.

As categorias de riscos propostas por Marty Cagan é uma boa forma de organizar e definir os critérios:

- **Usabilidade:** Há pontos de atrito? A API está intuitiva? É difícil cometer erros críticos? Segue os padrões conhecidos ou as melhores práticas? Como os devs se sentem ao interagir com o protótipo? Proporciona uma boa experiência?
- **Valor:** A solução resolve o problema certo, do jeito certo? É adequada ao contexto dos times usuários? Ajudará os times a resolverem problemas dos usuários finais? Favorecerá lançamentos no mercado em menor tempo (*time-to-market*)?
- **Viabilidade do negócio:** Está de acordo com normas, regulamentações ou leis? Qual o retorno para o negócio?
- **Viabilidade técnica:** É reutilizável e escalável? É possível manter os recursos necessários ao longo do tempo? É possível desenvolver dentro do prazo determinado?

Depois de analisar quais desses itens serão considerados no teste, definem-se *tarefas* relacionadas aos critérios.

Definição das tarefas

Você vai definir um roteiro com sequências de tarefas que o usuário terá de executar e completar para obter a resposta que você está buscando. Por exemplo, como a pessoa faria para gerar uma API key; o que é possível entender de determinado exemplo de código ou mensagem de erro; o que significa a nomenclatura de determinado atributo para ela; como ela faria para encontrar uma informação específica.

Definição de sucesso

Uma vez que os critérios e tarefas foram definidos, é preciso pensar nos dados qualitativos ou quantitativos que serão coletados para entender se a proposta do protótipo é bem-sucedida.

Vamos supor que uma das tarefas seja criar a API key e você quer saber se é fácil de usá-la. Alguns critérios de facilidade de uso poderiam ser o tempo para encontrar onde gerá-la e saber como aplicá-la no código. Você pode estabelecer um marco de sucesso *temporal*, como "até 1 minuto para encontrar onde gerar API key e gerá-la"; outra pode ser em *como fazer*, por exemplo "inserir a API key em determinado trecho de código corretamente". Essa é uma forma de você obter o seu ponto de vista.

Tão (ou mais) importante quanto o seu ponto de vista é o ponto de vista dos devs usuários. Então, depois do teste, você pode pedir às pessoas para classificarem o nível de facilidade para cumprir as tarefas, usando a escala *Likert*, em que elas classificam se foi *muito fácil*, *um pouco fácil*, *nem fácil nem difícil*, *um pouco difícil* ou *muito difícil*. Você pode usar qualquer ferramenta de formulário para isso ou até mesmo o papel. O *print* adiante exibe um exemplo de questão de múltipla escolha no Google Forms.

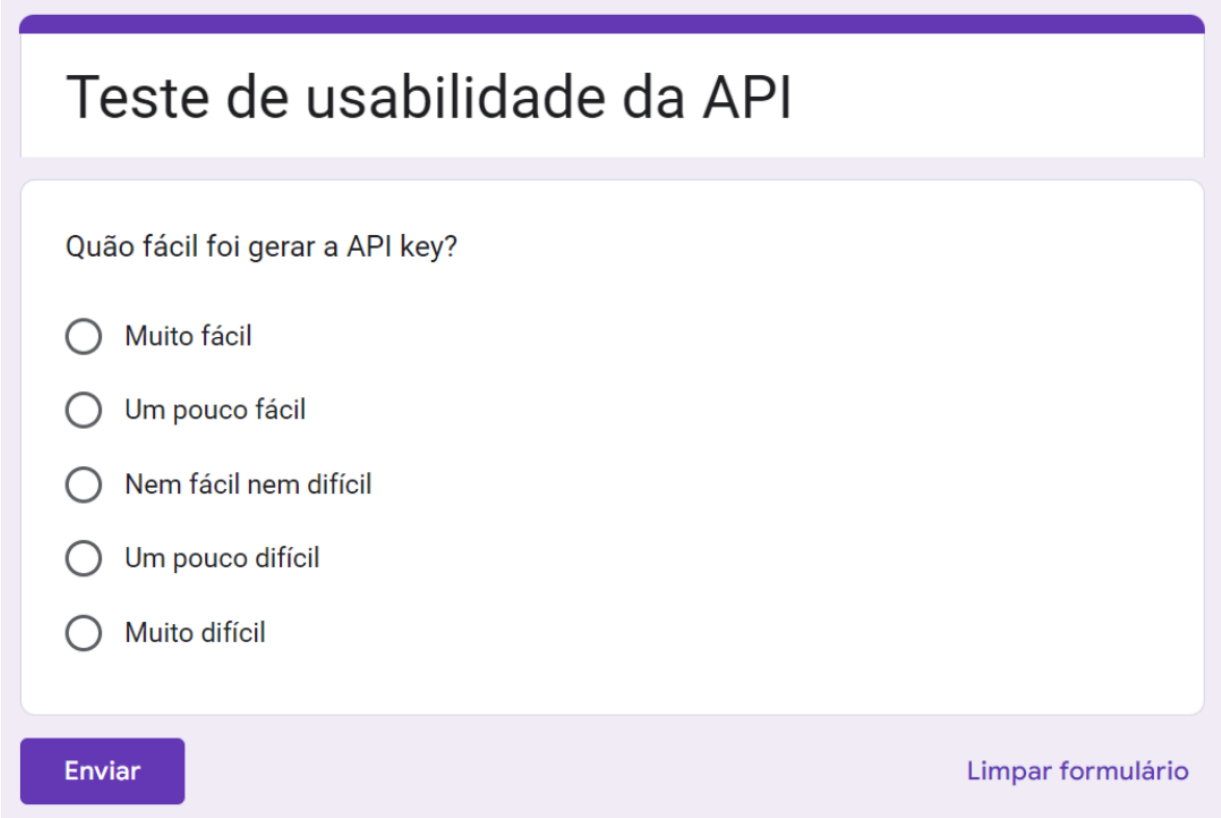


Figura 16.1: Questão de escala Likert.

Para essa escala, poderia ser determinado que a decisão de prosseguir com a implementação da solução prototipada atende aos critérios de usabilidade se a somatória das respostas "muito fácil", "um pouco fácil" e "nem fácil nem difícil" for superior à somatória das demais classificações.

Dependendo do tipo de pergunta ou afirmação, algumas escalas conhecidas vão de *concordo totalmente* a *discordo totalmente*, de *muita frequência* a *nunca*, de *muito importante* a *sem importância*. Mas na prática, pode ser qualquer coisa que faça sentido para o contexto, como de *muito bom* a *muito ruim* ou de *muito organizado* a *muito desorganizado*, por exemplo.

O resultado do feedback pode ser inconclusivo devido às variações de comportamento e respostas, ou porque a solução não cumpre todos os critérios. Esse é o sinal de que a solução ainda não é eficaz

o suficiente, precisa ser revisada e aprimorada antes de desenvolvê-la.

16.2 Como conduzir o teste de usabilidade

As sessões podem ser remotas ou presenciais, síncronas ou assíncronas. Em qualquer situação, o ideal é que os devs que participarão do teste utilizem os computadores deles para ficarem mais próximos das condições a que estão acostumados. Isso minimiza a necessidade de aprender como lidar com o ambiente desconhecido, evitando ruído na interpretação das informações do teste.

Em testes assíncronos, as pessoas participantes preenchem um formulário com questões específicas e você também pode fornecer espaços para comentários livres. Nas sessões síncronas, os usuários fornecem *feedback* em tempo real e você (ou outra pessoa do time) anota o que observa.

Nessas sessões, é importante a participação de pessoas-chaves para a coleta de feedback dos devs usuários: você, como PM, para entender o problema do negócio e avaliar a relevância da solução e, pelo menos, uma pessoa de engenharia — tech lead, designer da API ou software architect — que entenda a solução tecnicamente, as dependências e os impactos. Se o time não tem designer de API, seu papel será fundamental para interpretar a usabilidade sob o ponto de vista dos devs usuários e, provavelmente, você será também a pessoa facilitadora da sessão.

No início da sessão, explique como será o teste, o objetivo e deixe claro que não é a pessoa quem está sendo avaliada, é a solução que está em jogo e você quer saber o ponto de vista dela. Em seguida, você apresenta o cenário, ou caso de uso, e passa a tarefa a ser executada.

Ao longo do processo, você adiciona as questões que você elaborou no planejamento, por exemplo, "o que é possível entender desse código, o que faz e qual é o objetivo?". É comum que surjam perguntas que não estão no roteiro, tanto partindo de você como das pessoas do seu time que estão na sessão.

Observe como as pessoas interagem com o protótipo. Peça para que compartilhem os pensamentos e as dúvidas, o famoso "pensar em voz alta". Em vez de responder diretamente, ajude a pessoa a resolver. Por exemplo, imagine que a pessoa pergunte: *onde tenho que ir para gerar a API key?* Você devolve com a pergunta: *de onde você está, como você faria?* Se ela não conseguir sozinha, você responde. Aproveite para investigar por que a pessoa não conseguiu executar e, no ponto de vista dela, o que poderia ser diferente para que ela conseguisse. Se a pessoa diz que determinada nomenclatura soa estranha ou que ela entendeu algo diferente do que deveria, por exemplo, é importante você saber o motivo pelo qual a pessoa pensa isso e qual nome ela considera melhor.

Peça aos usuários para compararem o protótipo com outras soluções que eles conhecem. Isso pode revelar necessidades latentes. Ao final, se você criou algum formulário para estruturar a coleta dos *feedbacks*, peça aos participantes do teste para respondê-lo.

O objetivo primordial do *feedback* é validar se o problema foi entendido, se você conhece as tarefas que o dev usuário precisa realizar e se a solução atende às necessidades. Para isso, você pode perguntar:

- Qual a relevância desse cenário (ou caso de uso) para você?
- Quão bem a API resolveu o problema?

Essas perguntas validam a relevância de um cenário específico e a eficácia da solução proposta — o *problem-solution fit*.

16.3 Sintetização dos feedbacks

Depois dos testes, você vai analisar, sintetizar os *feedbacks* e chegará a uma conclusão. Nesse ponto, você já deve ter clareza sobre as limitações da plataforma e uma previsão de como os usuários se comportariam, pensariam e sentiriam ao interagir com a solução final.

A primeira coisa é validar o problema: o problema do cliente interno foi entendido? Depois, validar a solução: é a solução certa para resolver o problema? As respostas para essas perguntas são a evidência que você precisa para avaliar se vale a pena avançar para o desenvolvimento.

16.4 Iteração, pivô e adaptação

É difícil (para não dizer raro) que a solução seja eficaz, de primeira, para o contexto de todos os times usuários. Os testes servem para a identificar elementos e variantes que funcionam bem e também o que não funciona ou nem deveria existir, além de revelar novos insights.

Lembra que o Design Thinking é uma atividade iterativa? A compreensão mais robusta do que os devs usuários podem enfrentar ao interagir com a solução em determinado contexto retroalimenta o processo, redefinindo o que foi estabelecido em etapas anteriores.

Assim, pode-se voltar para a *Prototipagem*, quando se faz simples alterações em algumas partes do protótipo até que demonstre ser satisfatório para o desenvolvimento; ou o total descarte da solução e criação de novos protótipos, o que leva de volta para o estágio de *Ideação*.

Às vezes, o teste mostra que não apenas se errou na solução, mas que o problema foi mal-entendido no começo ou ele não foi enquadrado adequadamente, por isso é necessário refiná-lo. Talvez o problema seja completamente diferente do que havia se imaginado. Nessas situações, retorna-se para o estágio *Definição*.

O entregável do processo de Design Thinking é a decisão de avançar para a construção (ou não) da solução fundamentada em evidências. Para isso, há inúmeros exercícios para cada estágio do Design Thinking que podem ser usados como você quiser, com total liberdade para aprimorar o processo ao que funciona para você, como menciona o Hasso Plattner Institute of Design at Stanford (c2023): "*em última análise, você fará o seu próprio processo e vai adaptá-lo ao seu estilo e trabalho*" (tradução livre).

No último capítulo desta parte do livro exploraremos a aplicação prática do Design Thinking na criação de APIs. Mas antes disso, que tal aprender a testar as APIs que o seu time cria? O próximo capítulo ensina como fazer isso.

Capítulo 17
Como PMs conseguem testar APIs

Sabe aquele dito popular "de médico e louco todo mundo tem um pouco"? Eu refrasearia para "de médico, louco e *designer* todo mundo tem um pouco". As pessoas veem, interagem com design de interface visual (UI) e conseguem, ao menos, ter uma opinião do tipo "gosto" ou "não gosto". Como elas acreditam que entendem de design (pautadas em seus gostos), elas distribuem seus pitacos. PMs podem facilmente cair na tentação de interferir no design, propondo ou até impondo suas ideias.

Já quando o assunto é *design de API*, não se pode dizer o mesmo. A pessoa PM mal consegue entender, muito menos emitir opiniões — a não ser que ela tenha *background* técnico.

Neste capítulo, vamos nos aprofundar ao que foi abordado em linhas gerais no capítulo *Anatomia da plataforma*, o funcionamento de API síncrona. O objetivo é que você consiga testar as APIs que seu time desenvolve e possa compreender o *techniquês* para se comunicar melhor com as pessoas de engenharia, elaborar e conduzir entrevistas com este público.

Segundo a pesquisa *2022 State of the API Report* feita pelo Postman (2022), REST é, disparada, a arquitetura de API mais usada (89%) — em segundo lugar encontra-se Webhooks, com 35%. Por isso, REST é o estilo que estudaremos aqui.

APIs REST usam o protocolo HTTP, a comunicação padrão da internet. Relembrando, API é uma interface de comunicação entre sistemas em que o sistema cliente (consumidor da API) envia requisições ao sistema servidor e aguarda a resposta. O sistema servidor processa a informação e/ou realiza cálculos dessa entrada

e, então, retorna os dados de saída ao sistema cliente. Para que isso aconteça, a requisição deve seguir um modelo específico e ser enviada a um endereço pelo qual o sistema servidor recebe e responde à requisição.

Veja essa analogia (clássica) que compara o funcionamento de APIs à operação de restaurantes — inspirado na didática do vídeo *All you need to know about APIs* (2022).

17.1 APIs funcionam como restaurantes

Imagine que você vai a um restaurante e olha o cardápio, que está organizado em seções: "pizzas" e "esfihas". As seções possuem uma lista de itens (os sabores) e o respectivo preço ao lado de cada um deles. Só é possível pedir o que a cozinha tem para fornecer, ou seja, você não pode pedir feijoada, uma vez que o restaurante não vende esse tipo de refeição.

Você chama o garçom para fazer o pedido e diz a ele: "muçarela". O garçom fica parado, olhando para você, até que você completa: "pizza, uma pizza de muçarela". Ao dizer apenas "muçarela", o garçom não sabe se é pizza ou esfiha. Você tem que especificar o que você deseja e o sabor — *pizza* de *muçarela*. Somente "pizza" ou somente "muçarela" não comunica com precisão o que você quer. É importante mencionar da mesma forma em que aparece no cardápio. Se, na verdade, você quer "pizza de muçarela à moda da casa", mas disser "pizza de muçarela", você vai receber a pizza de muçarela, mas não à moda da casa. Você também precisa dizer a quantidade, porque sem essa informação o garçom pode trazer quantas pizzas ele quiser.

É preciso seguir regras para que o pedido seja bem-sucedido: pedir pelo nome igualzinho ao que está no cardápio e especificar um parâmetro *obrigatório*, a quantidade. O garçom anota o seu pedido e

leva para a cozinha. Na cozinha, a pizza é preparada. Depois de pronta, a pizza é entregue ao garçom, que, por fim, entrega a pizza a você. Traduzindo tudo isso para o mundo das APIs:

- Garçom: é a **API**, uma intermediária que sabe como receber e fornecer os dados;
- Você: é o **sistema cliente**, que consome a API e deseja obter dados;
- Cardápio: é a **documentação**, que explica o que pode ser solicitado à API, o formato de dados que os sistemas externos devem aplicar para executar uma operação;
- Cozinha: é o **sistema servidor** com um banco de dados e, para proteger a integridade dos dados, nenhum cliente tem permissão de acesso;
- Pedido: é a **requisição** (*request*), também conhecida como **solicitação** ou **chamada**;
- Pizza de muçarela: é a **resposta** (*response*) à requisição;
- Regras: é o **contrato**, um padrão que determina como as informações podem ser transferidas para que os dois sistemas consigam interagir.

Por que não fazer a comunicação direta em vez de via API? Primeiro, para manter os sistemas independentes. Se você tivesse permissão para entrar na cozinha, ia sobrar louça para você lavar, e não é o que você quer, né? Segundo, como você se comunicaria com o garçom se não tivesse visto o cardápio? Se você disser "eu quero pizza", o garçom vai entender o que você quer dizer, mas, para entregar a coisa certa, ele precisa de mais detalhes de acordo com o **contrato**. Observe como é isso com uma API real.

17.2 Como é e como funciona uma API real

Para conhecer a aparência e como funciona uma API REST, vamos usar uma das APIs do Google Maps, a API Places

(https://developers.google.com/maps/documentation/places/web-service).

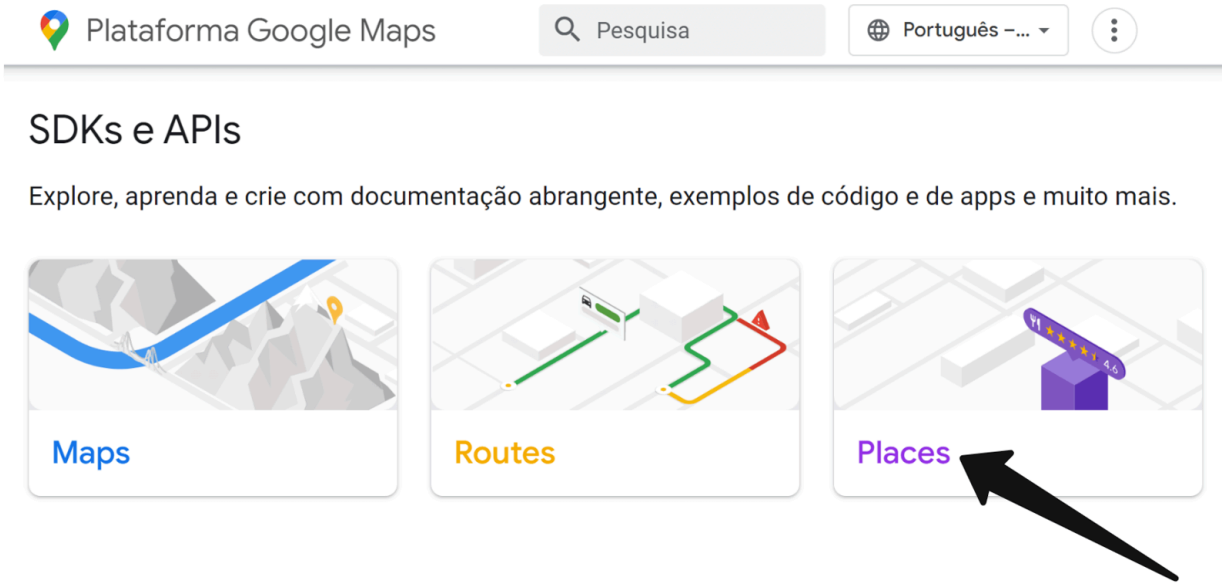


Figura 17.1: Portal do desenvolvedor. Fonte: Plataforma Google Maps.

A página "Visão geral" do portal do desenvolvedor da API Places exibe os casos de uso, como funciona, como usar e a lista de recursos disponíveis. Vamos pegar como exemplo a "Pesquisa de local":

Figura 17.2: API Places. Fonte: Plataforma Google Maps.

Entre outras coisas, a página "Pesquisa de local" mostra os tipos de pesquisa que podem ser realizadas. Na próxima imagem é possível saber que "*há três endpoints*": "Find place", "Pesquisa de proximidade" (*Nearby search*) e "Pesquisa de texto" (*Text search*) — repare que alguns termos foram traduzidos para o português e outros não, o que causa um probleminha de usabilidade no quesito *consistência* em demais partes da documentação em que foi mantido em inglês. Mas o que é *endpoint*?

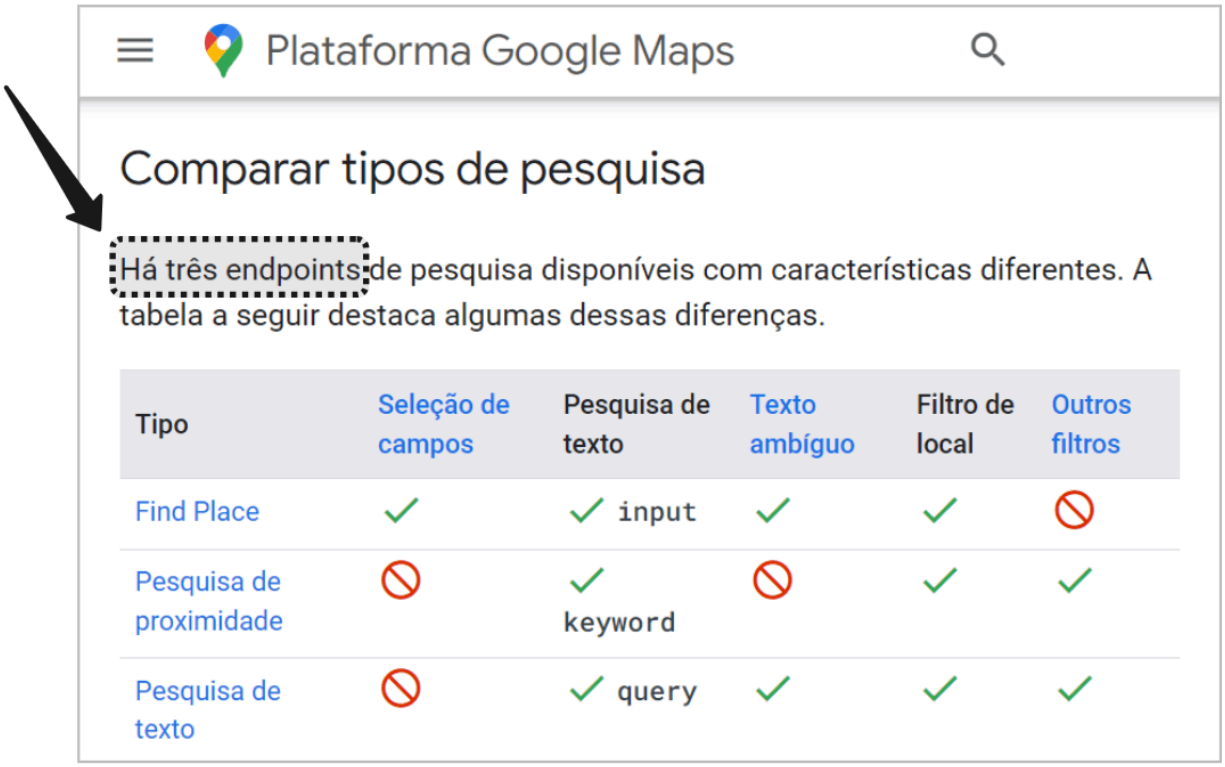


Figura 17.3: API Places — Endpoints. Fonte: Plataforma Google Maps.

Endpoint

Endpoint é o ponto de contato entre os sistemas cliente e servidor. A API Places disponibiliza três pontos pelos quais a API recebe a requisição, para então acessar os recursos da "Pesquisa de local" no back-end, e devolver a resposta.

Imagine uma casa: ela é a plataforma e a porta de entrada é a API. Uma vez que você passa pela porta, você se depara com os cômodos da casa e cada um deles possui recursos diferentes. Por exemplo, se você quiser comer alguma coisa, você terá que passar pela porta da cozinha, mas se você quiser dormir, você precisa acessar a porta do quarto. A necessidade ou requisito é o que define o cômodo que você vai acessar, embora dentro da mesma casa. Resumindo, a API é a porta principal da casa e cada porta dentro dela é um *endpoint* que dá acesso a um cômodo, onde se encontra um recurso específico.

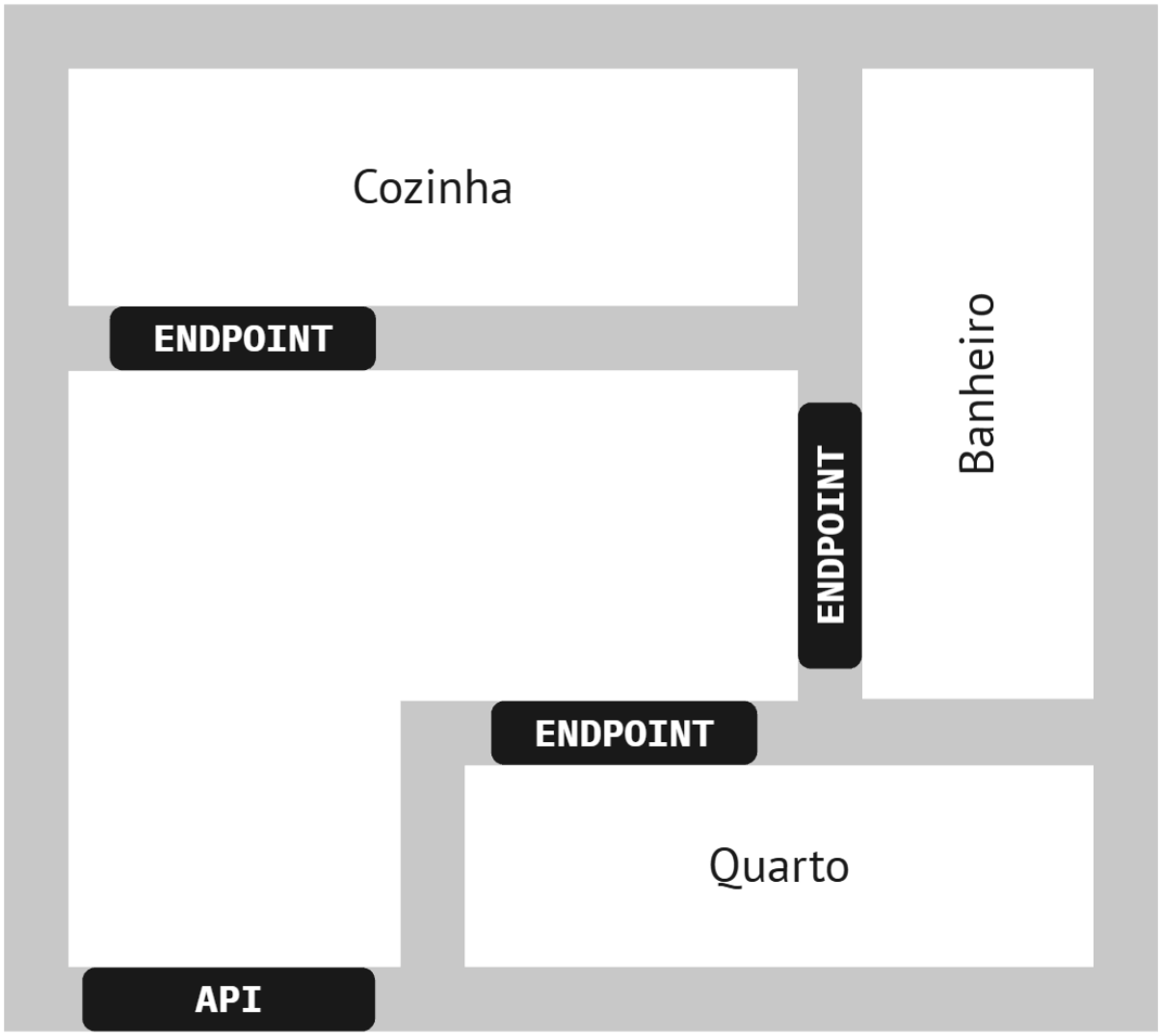


Figura 17.4: API e Endpoints.

De volta ao Google Maps, a página do *endpoint* "Pesquisa de texto" diz que, dada uma *string* (texto), como "pizza em Nova York" ou "lojas de sapatos perto de Ottawa", esse *endpoint* retorna uma lista de locais.

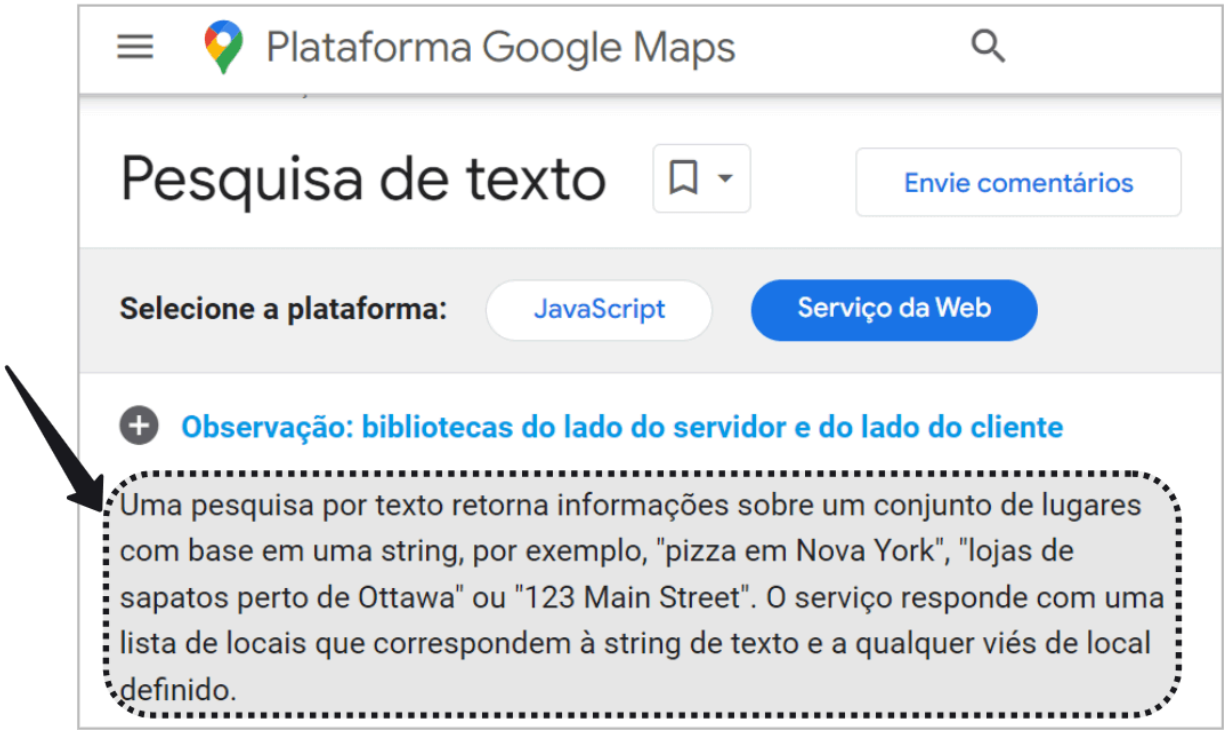


Figura 17.5: Fonte: Plataforma Google Maps.

Para funcionar, a requisição deve seguir os padrões determinados na documentação. Primeiro, é preciso acessar o *endpoint*, e isso é feito por meio de um endereço na internet, a URL, como pode ser conferido no *print* a seguir.

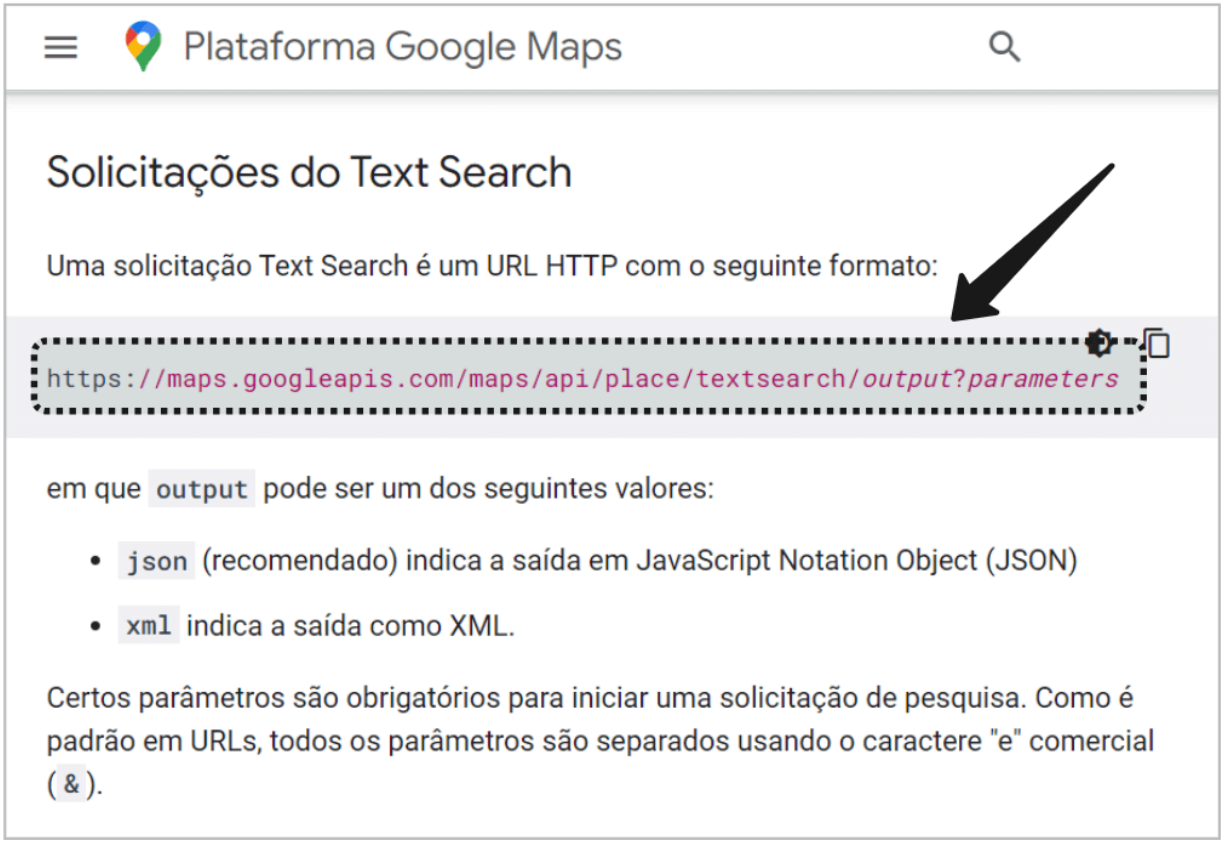


Figura 17.6: Fonte: Plataforma Google Maps.

URL

Assim como você digita uma URL no navegador para visitar uma página da internet, você precisa de uma URL para acessar determinada API. Para acessar o *endpoint* "Pesquisa de texto" (*TextSearch*), a documentação indica o seguinte endereço:

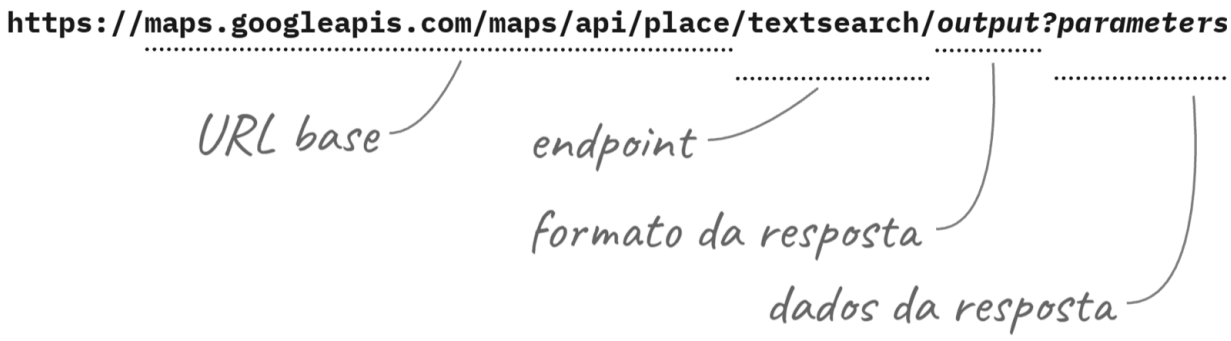


Figura 17.7: URL da API Places.

`https://maps.googleapis.com/maps/api/place/` é a URL base, a porta da casa. O cômodo "Pesquisa de texto" é acessado pela porta (*endpoint*) `textsearch` . Supondo que se queira acessar o *endpoint* "*Find Place*", o valor `textsearch` deve ser substituído na URL por `findplacefromtext` ; e quando você quiser acessar "Pesquisa de proximidade", você tem que substituir por `nearbysearch` .

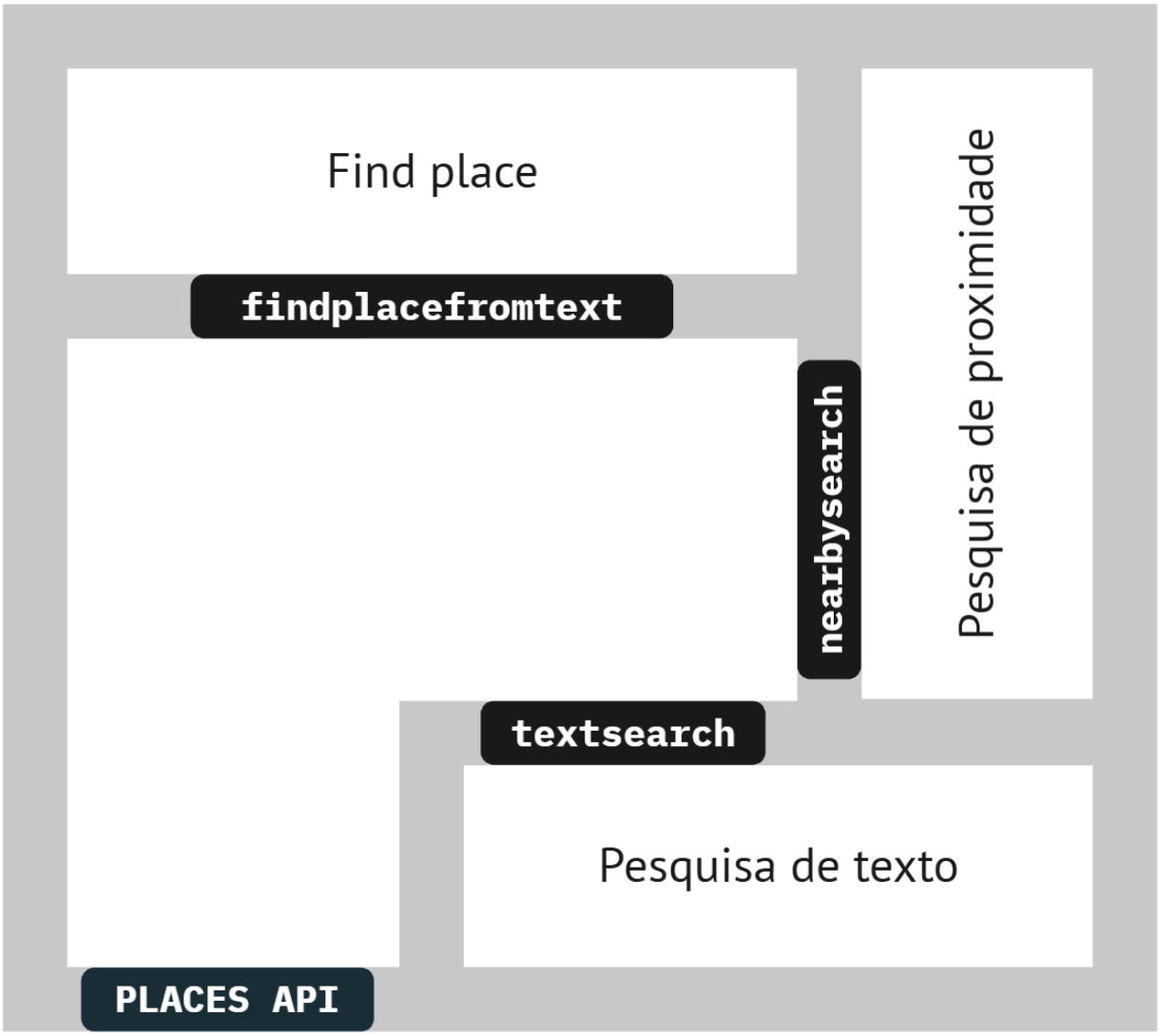


Figura 17.8: API e Endpoints.

O `output` refere-se à linguagem na qual o dev usuário da API quer receber a resposta. Confira na imagem o exemplo para quando se quer a resposta em JSON.

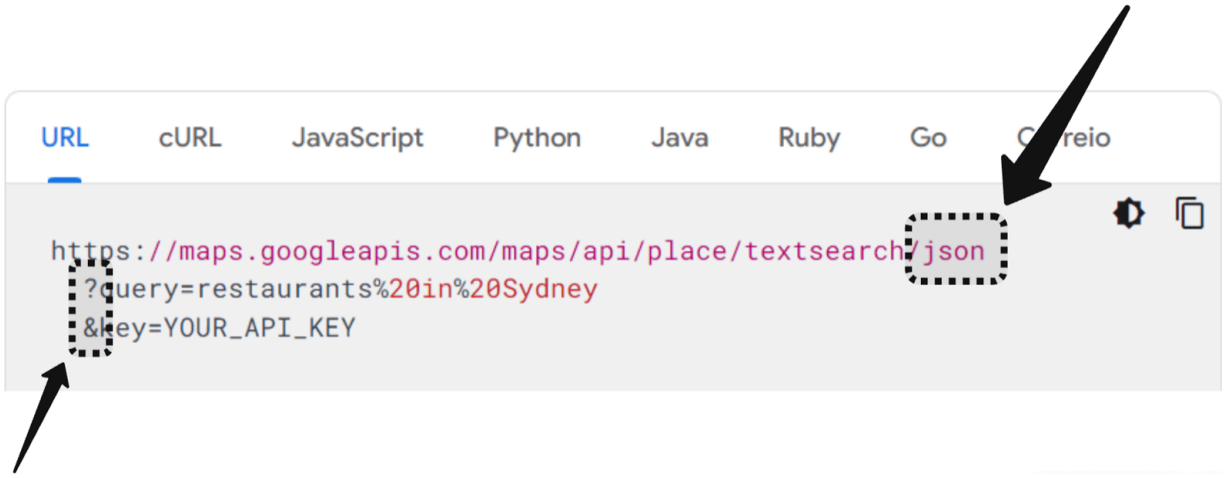


Figura 17.9: Fonte: Plataforma Google Maps.

Repare na figura anterior que, depois de *output*, há o sinal de interrogação (?). Esse sinal indica que o que vem depois dele são **parâmetros**, "*frequentemente usados em requisições HTTP para filtrar respostas ou fornecer informações adicionais sobre a requisição*" (HUNTER, 2017; tradução livre).

Parâmetros

A interrogação (?) separa a URL dos parâmetros, e cada par *chave-valor* é separado pelo sinal de igual (=) — `chave=valor`. A figura mais adiante mostra o exemplo de código para busca por restaurantes em Sydney, em que os parâmetros são:

- `query=restaurants%20in%20Sydney`: `query` é o tipo de parâmetro, no caso, o termo de pesquisa em formato texto — `%20` é a forma de dizer às máquinas que há espaço entre as palavras (você não precisa se preocupar com isso);
- `key=YOUR_API_KEY`: voltando ao exemplo da casa, se você quer entrar nela, você precisa de uma chave. Isso significa que não é permitido o acesso a essa API sem uma API key.

Note que há dois tipos de parâmetros e, para separar um do outro, é usado "&" (e comercial).

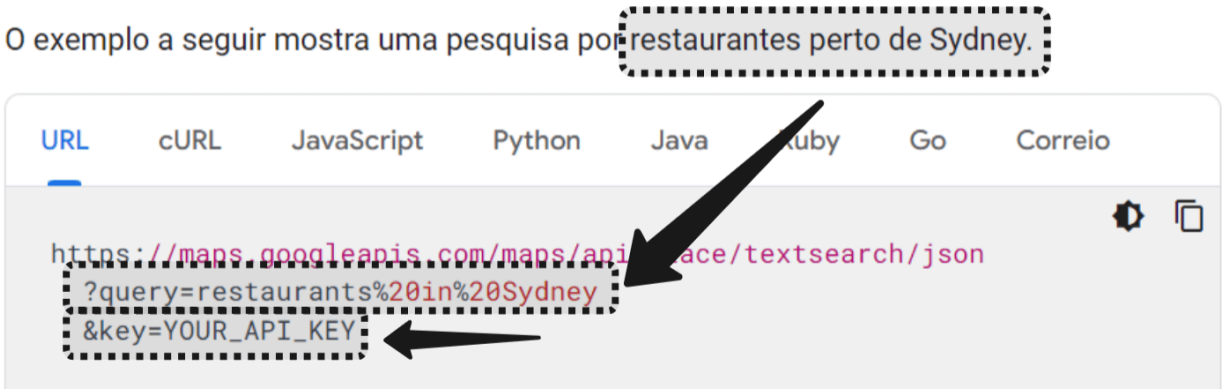


Figura 17.10: Fonte: Plataforma Google Maps.

Resposta

A imagem a seguir apresenta a resposta em JSON à requisição "*restaurants in Sydney*". Esse *endpoint* também responde em XML, mas é possível fornecer a resposta em outros formatos, como CSV e texto simples, por exemplo.

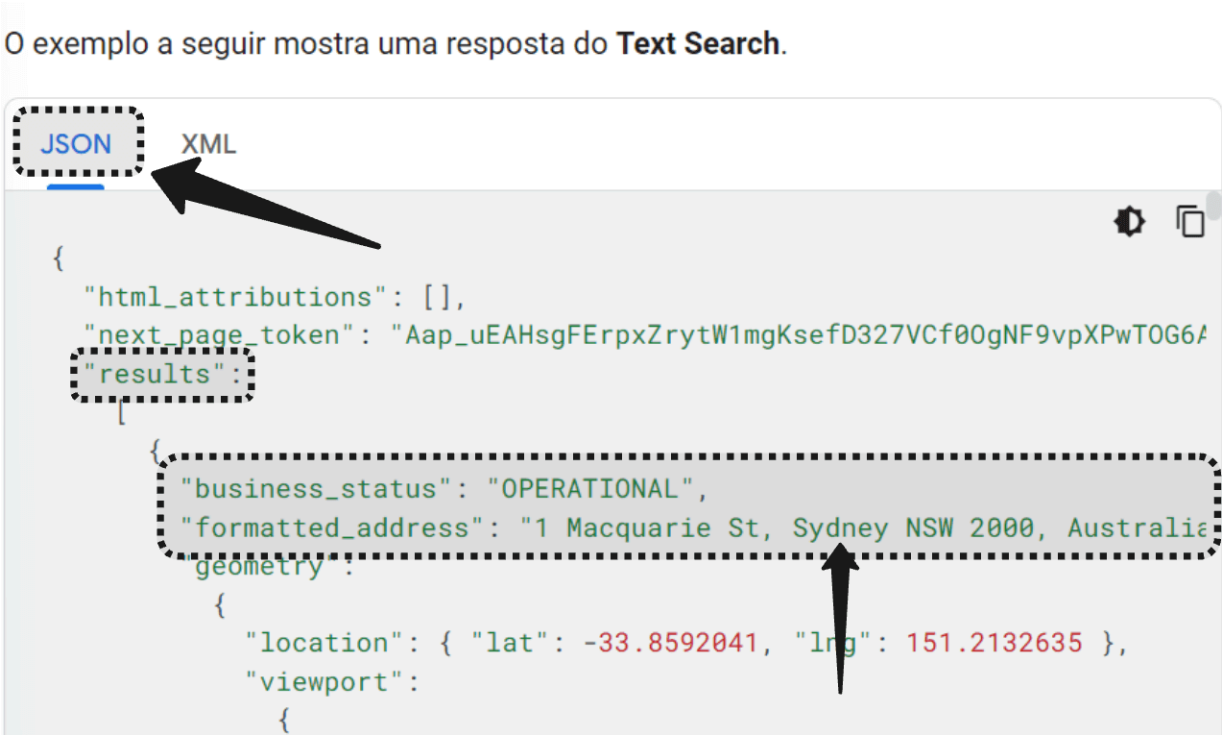


Figura 17.11: Fonte: Plataforma Google Maps.

Se esse código é intimidador e você acha difícil de entender, observe o formulário a seguir. Ele segue uma hierarquia em que os dados são encadeados uns "dentro" de outros.

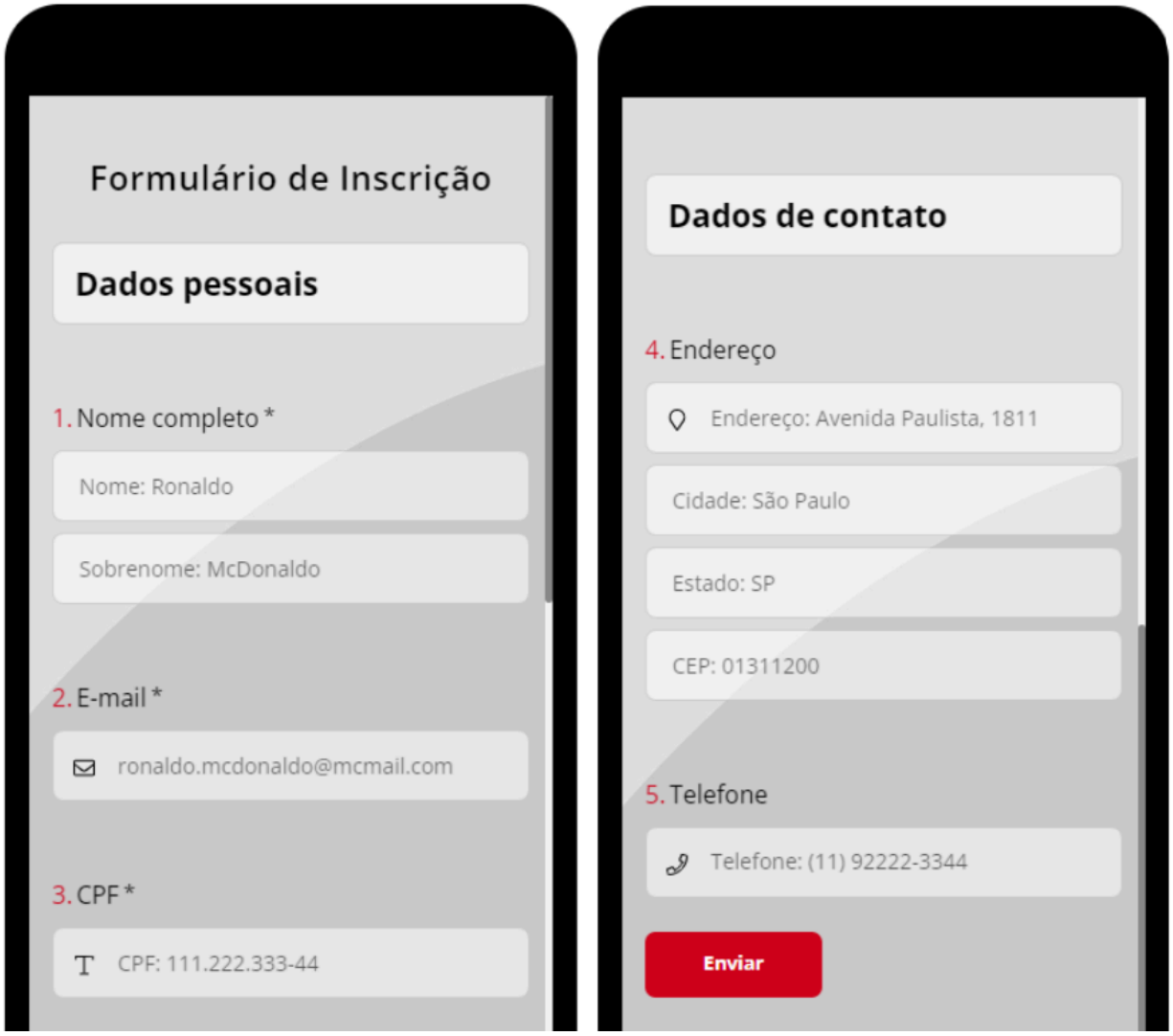


Figura 17.12: Formulário.

Adiante você vê como seria esse formulário em formato texto. Percebe-se dois grupos: "Dados pessoais" e "Dados de contato". Ambos estão contidos em "Formulário de Inscrição", portanto estão um nível abaixo na hierarquia da informação. Repare que os dados de "Nome completo" e "Endereço completo" são segregados e possuem nomenclaturas específicas — o encadeamento dos *bullets* deixa evidente que pertencem ao conjunto de nível superior.

Formulário de Inscrição

Dados pessoais

- Nome completo
    - Nome: *Ronaldo*
    - Sobrenome: *McDonaldo*
- E-mail: *ronaldo.mcdonaldo@mcmail.com*
- CPF: *111.222.333-44*

Dados de contato

- Endereço completo
    - Endereço: *Avenida Paulista, 1811*
    - Cidade: *São Paulo*
    - Estado: *SP*
    - CEP: *01311200*
- Telefone: *(11) 92222-3344*

Figura 17.13: Formulário em formato texto.

Agora, se substituirmos os *bullets* por chaves que delimitam o conjunto ao qual os dados pertencem, como mostra a imagem seguinte, o resultado é semelhante ao código em JSON. A *indentação* (distância da margem) indica a hierarquia, e as chaves, o conjunto de dados pertencentes à informação de nível superior.

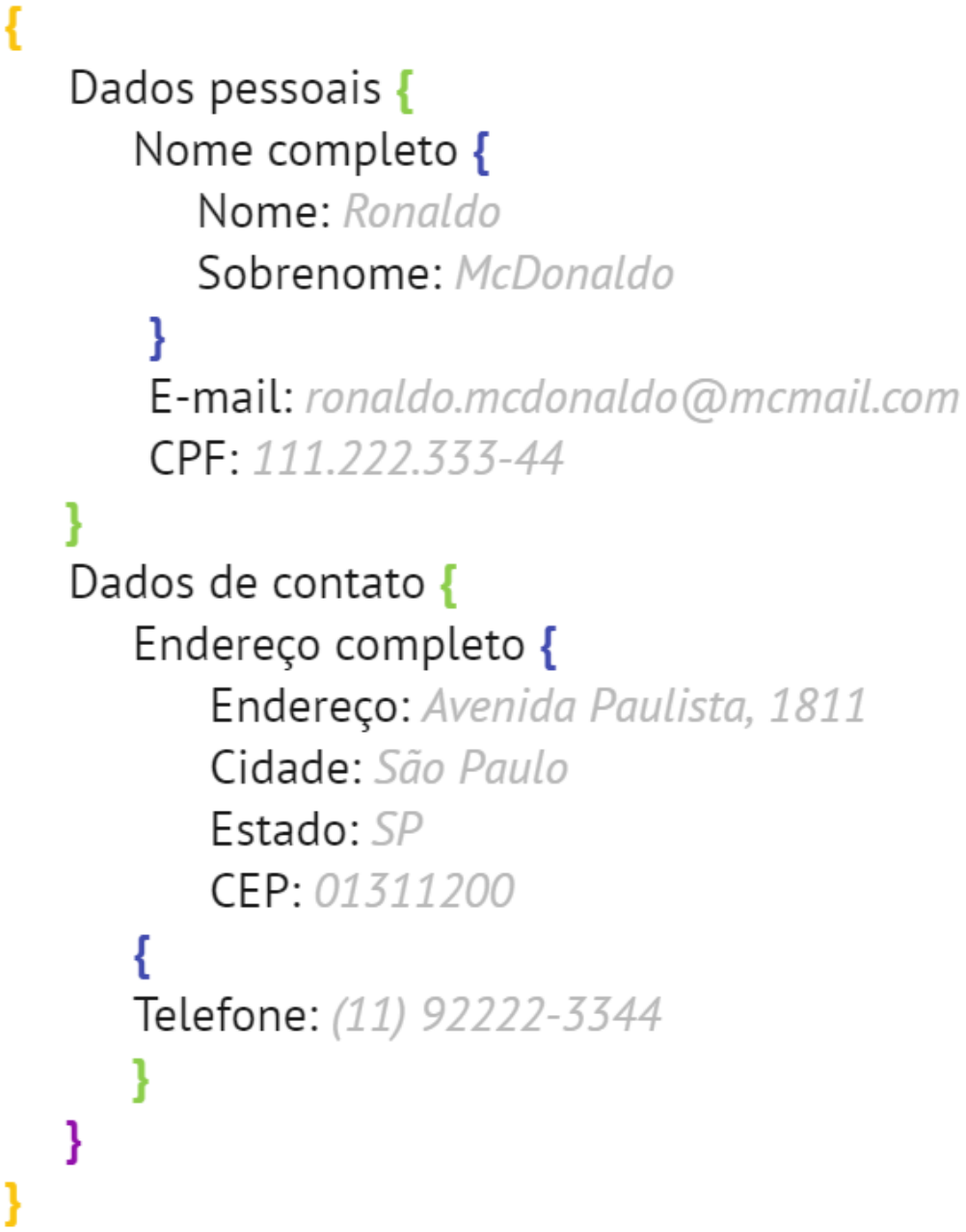
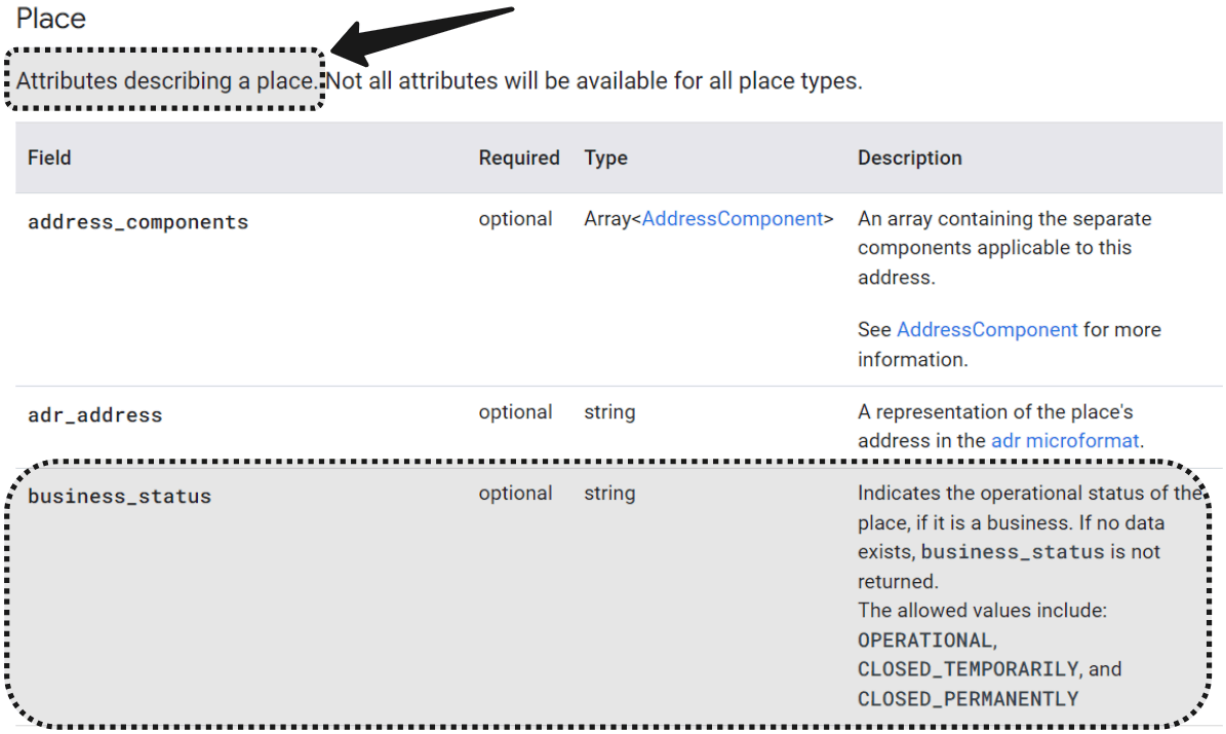


Figura 17.14: Formulário em formato texto.

Voltando à API Places (figura 17.11), a informação relevante está dentro de `results` (resultados). O primeiro atributo de um dos resultados é `"business_status": "OPERATIONAL"`, que se refere ao status de negócio. Como é possível saber isso? A documentação explica o significado de cada dado:



Place

Attributes describing a place. Not all attributes will be available for all place types.

Field	Required	Type	Description
`address_components`	optional	Array<AddressComponent>	An array containing the separate components applicable to this address. See AddressComponent for more information.
`adr_address`	optional	string	A representation of the place's address in the adr microformat.
`business_status`	optional	string	Indicates the operational status of the place, if it is a business. If no data exists, `business_status` is not returned. The allowed values include: `OPERATIONAL`, `CLOSED_TEMPORARILY`, and `CLOSED_PERMANENTLY`

Figura 17.15: Fonte: Plataforma Google Maps.

O próximo atributo é `formatted_address`, o endereço exato do local. Já na próxima imagem vemos, `"name": "Aria Restaurant Sydney"`, o nome do restaurante e que esse restaurante tem fotos publicadas no Google Maps sob o atributo `photos`, e o `"open_now": false` indica que o estabelecimento não estava aberto no momento da criação desse exemplo de código:

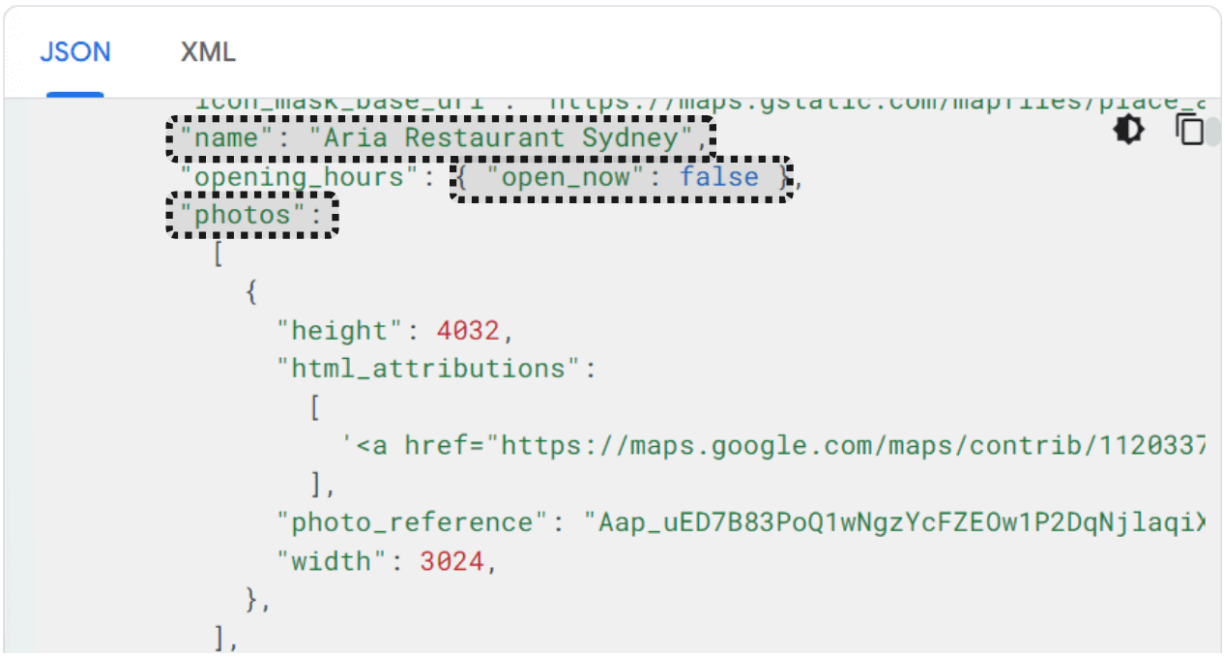


Figura 17.16: Fonte: Plataforma Google Maps.

Mais adiante no código tem a classificação (`"rating": 4.5`) e o total de pessoas que classificaram o Aria Restaurant Sydney (`"user_ratings_total": 1681`).

Figura 17.17: Fonte: Plataforma Google Maps.

Quando alguém digita um termo no Google Maps é enviada uma requisição como a que vimos e a pessoa obtém uma lista com vários restaurantes. Vamos conferir se o Aria Restaurant Sydney aparece no resultado para a busca "restaurants near sydney" — nesse exemplo, precisa ser em inglês; em português, o comportamento pode ser diferente, devido a limitações do sistema tradutor.

A resposta é exibida na interface em forma de lista de restaurantes e os *pins* que apontam a localização deles no mapa. o Aria Restaurant Sydney é o segundo resultado, onde exibe a foto, a avaliação (4,5) e o número de pessoas que avaliaram (1.838) — o número de avaliações está diferente do código que vimos porque o exemplo da documentação não é em tempo real. Veja que o endereço se confirma e que está fora do horário de funcionamento (no momento do *print* da imagem).

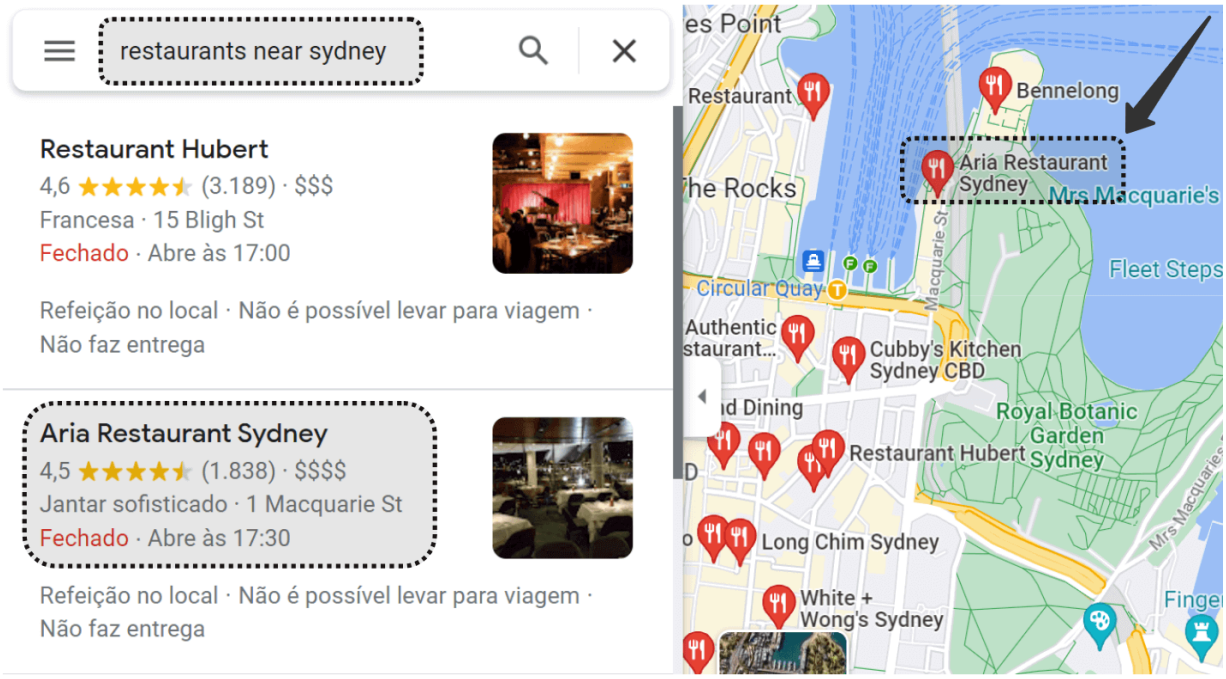


Figura 17.18: Fonte: Plataforma Google Maps.

Será que o primeiro restaurante do resultado — Restaurant Hubert — aparece no exemplo de código de resposta na documentação? Rolando o código, lá está, `"name": "Restaurant Hubert"`.

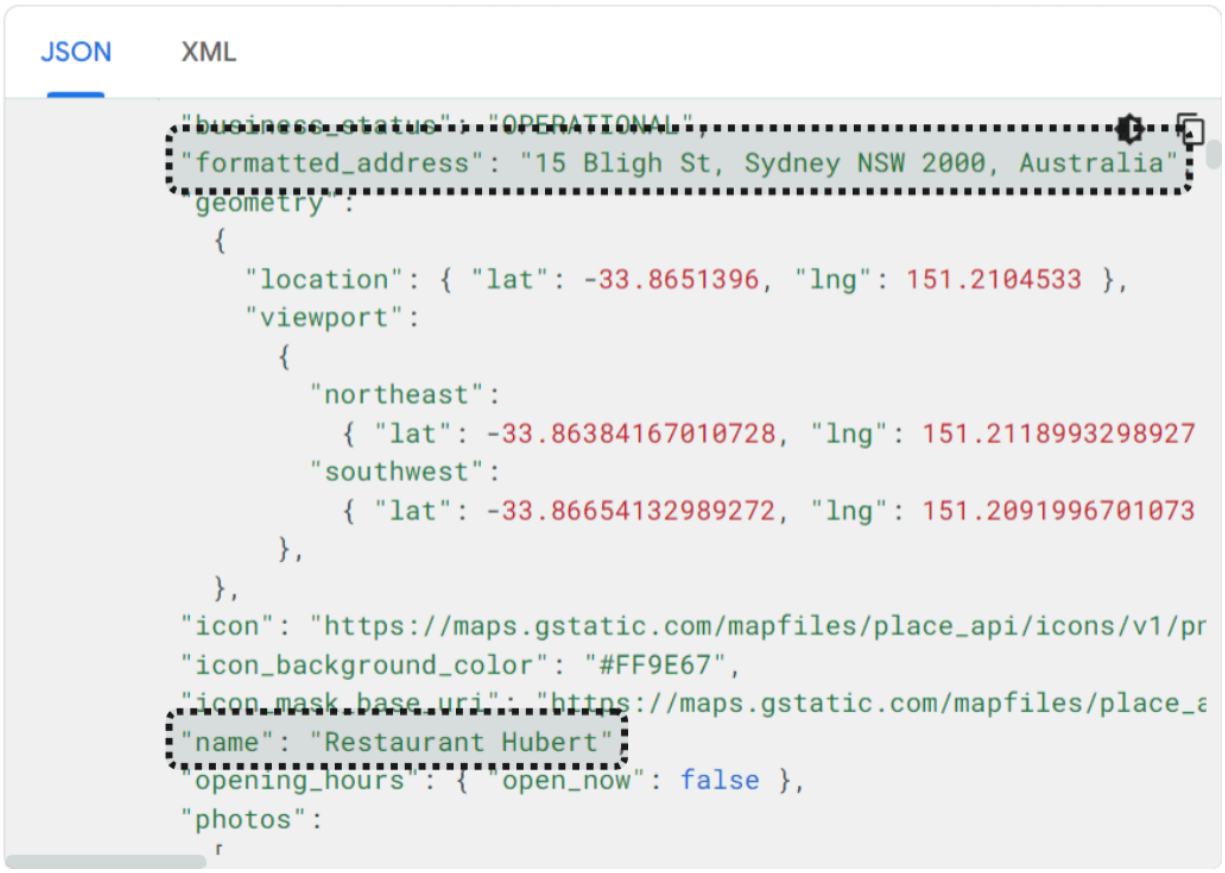


Figura 17.19: Fonte: Plataforma Google Maps.

É assim que a API Places do Google Maps funciona. Toda vez que uma pessoa pesquisa, ela está requisitando algo à API e recebe uma determinada resposta de volta. Claro que seria inconveniente o usuário final inserir no navegador a URL com os parâmetros, então o sistema cliente faz isso pela pessoa: interpreta o termo a ser buscado e o transforma em algo que o sistema servidor vai entender. O mesmo ocorre com a resposta. O sistema cliente entende em formato "bruto" e o transforma em UI para o usuário final, por meio de HTML, CSS e Javascript.

O exemplo de código de resposta apresenta uma lista com vários restaurantes e, no final dela, o *status* à requisição:

Figura 17.20: Fonte: Plataforma Google Maps.

Status

Assim como toda requisição deve ter um endereço (URL), toda resposta deve ter o *status* que indica sucesso, erro ou falha à requisição. O `"status": "OK"` presente no código da API do Google indica que tudo ocorreu bem. No caso, o *payload* traz apenas a mensagem (`"OK"`), mas é comum que ela venha acompanhada de um código numérico.

Os códigos e seus significados são padronizados, basta conhecê-los para oferecer suporte ou providenciar correções. Existem cinco tipos de códigos de status:

- `1xx` : de 100 a 199 referem-se a respostas de informação;
- `2xx` : de 200 a 299 são para **respostas de sucesso**;

- `3xx` : de 300 a 399 para apontar redirecionamentos;
- `4xx` : de 400 a 499 são relacionados a **erros do sistema cliente**, para quando a requisição está fora do padrão do contrato e, por isso, não é compreendida pela API;
- `5xx` : de 500 a 599 são sobre **erros do sistema servidor**.

Retomando a analogia do restaurante, suponha que, ao mencionar "pizza de muçarela", você esqueceu de dizer a quantidade. Nesta situação, o garçom diria: "essa é uma requisição ruim" (*400 bad request*). Agora, imagine que o seu pedido foi bom, mas o restaurante ficou sem energia elétrica ou faltam ingredientes, portanto não é possível preparar a pizza que você quer. Você receberia um erro do tipo *500 Internal Server Error* (erro interno do servidor).

Você consegue encontrar facilmente o significado de cada código na internet. Esses estão entre os mais comuns, que de acordo com a Wikipédia (c2023) significam:

RESPOSTAS DE SUCESSO

Código	Mensagem	Descrição
200	OK	Requisição bem-sucedida. O significado de sucesso varia de acordo com o método HTTP — `GET` , `POST` , `PUT` , `DELETE` etc.
201	Created	O servidor confirma que o recurso foi criado. Geralmente, é a resposta à requisição `POST` .
202	Accepted	A requisição foi recebida pelo sistema servidor, mas o processamento ainda não foi completado, como ocorre em comunicação assíncrona.

ERROS DO SISTEMA CLIENTE

Código	Mensagem	Descrição
400	Bad Request	Requisição com dados incompletos, dados malformados (*malformed request*), sintaxe ou dados inválidos.
401	Unauthorized	É necessária a autenticação, mas ela não foi fornecida ou não possui as credenciais de autenticação válidas para acessar o recurso solicitado.
403	Forbidden	O sistema cliente foi reconhecido, mas o servidor recusa a ação porque o cliente não tem permissão ou tentou uma ação proibida.
404	Not found	O servidor está acessível, mas o recurso específico não existe.
405	Method not allowed	O método não é suportado pela requisição, por exemplo, uma requisição `PUT` (criar) para um recurso somente leitura. Este é um caso típico em que se recomenda ao dev usuário verificar a documentação para usar o método HTTP correto.

ERROS DO SISTEMA SERVIDOR

Código	Mensagem	Descrição
500	Internal Server Error	É uma mensagem genérica para quando ocorre algo inesperado e não

Código	Mensagem	Descrição
		há uma mensagem específica adequada.
503	Service Unavailable	O sistema servidor não consegue responder à requisição porque está inativo ou sobrecarregado.

Imagine uma situação em que o dev usuário recebe o erro "*403 Forbidden*" — a autenticação ocorreu com sucesso, mas algo na requisição foi proibida. Pode ser um tanto frustrante obter esse tipo de erro sem entender o *porquê* de ter sido proibido. Saiba que é possível deixar a mensagem amigável (*dev friendly*) para algo como: *Você não tem crédito suficiente para concluir esta transação*. Dessa forma, o dev sabe o que fazer para obter sucesso.

Métodos

Cada requisição precisa indicar a ação que o sistema servidor deve executar, como **criar** um novo registro, **ler** (listar ou consultar) registros, **atualizar** (alterar) informações e **excluir** registros. Esses verbos representam as quatro operações básicas, conhecidas como **CRUD** (*create*, *read*, *update*, *delete*), e que são representadas por **métodos HTTP**, como mostra a tabela a seguir, comparados com operações SQL que você já deve conhecer:

Operação	SQL	Método HTTP
Create	INSERT	POST
Read	SELECT	GET
Update	UPDATE	PUT
Delete	DELETE	DELETE

Vimos a URL para a pesquisa por restaurantes em Sydney, agora veja na próxima imagem o exemplo de código com o método para listar esses restaurantes em cURL e JavaScript:

Figura 17.21: Fonte: Plataforma Google Maps.

Este é um caso de consulta (leitura), por isso deve-se usar o método `GET` na requisição. Agora, vamos supor que você queira cadastrar um restaurante para ser exibido no mapa, ou publicar uma foto no Instagram, ou um comentário no Facebook. Nestes casos, você não quer obter uma lista com os registros, você quer criar um registro, a ser salvo no banco de dados. Para isso, a requisição deve ser `POST`, como apresenta a imagem seguinte da API Maps Datasets, usada para criar um conjunto de dados de local, como nome e endereço a ser exibido no mapa.

Figura 17.22: Fonte: Plataforma Google Maps.

E, como é de se esperar, se você quiser excluir um local, basta enviar uma requisição DELETE, assim como a próxima imagem.

Figura 17.23: Fonte: Plataforma Google Maps.

Contrato

Como vimos, as APIs têm regras sobre como interagir com uma plataforma e quais respostas esperar — o contrato que tanto o cliente quanto o servidor devem seguir para garantir uma comunicação eficaz. Este contrato descreve as especificações técnicas necessárias para que, quando mudanças internas ocorrerem, nenhuma das partes precise se ajustar para a outra.

O contrato contempla tudo o que vimos até agora: os detalhes básicos de como as requisições devem ser formatadas, estruturadas e validadas; o que se obtém de resposta desta interface; o que é necessário para acessar a interface; o que acontece se algo der errado; o formato de dados aceitos e retornados; os parâmetros permitidos; a codificação usada; os métodos de autenticação e segurança, entre outros detalhes técnicos.

Requisições sempre ocorrem por meio de uma URL e de um método que indica a ação que se pretende executar. Elas são compostas por *request header* (cabeçalho da requisição) e, dependendo do método, possuem *request body* (corpo da requisição) e/ou parâmetros anexados à URL. A API Places define no contrato que, para que a busca por um local ocorra com sucesso, a requisição deve ser do tipo `GET` e é necessário passar os parâmetros `query` e `key` anexados à URL — ainda será exposto como é uma requisição quando exige conteúdo no *body*.

Além do *status code* (código do status), respostas também são compostas por cabeçalho e corpo — *response header* e *response body*. O *body* contém o conteúdo da resposta — dados solicitados pelo sistema cliente ou mensagem de erro.

Em ambas as situações, o *header* fornece informações adicionais. Por exemplo, `Content-Type: application/json` indica o formato do conteúdo do corpo *body* da requisição (JSON, nesse caso); `Allow: POST, GET` são os métodos permitidos em determinado *endpoint*, retornado quando o cliente envia uma requisição que resulta no erro `405 Method Not Allowed`.

O conceito de contrato ficará mais claro nos testes a seguir.

17.3 Teste na prática

Finalmente, chegamos ao tão esperado momento de experimentar APIs e entender como é a experiência do dev usuário. Para isso, vamos manusear o Postman, a versão web (https://www.postman.com/). Você pode fazer o download, se preferir.

Primeiro, você precisa criar uma conta, se ainda não tiver. Após o login, você vai se deparar com um *workspace* em branco como do *print*:

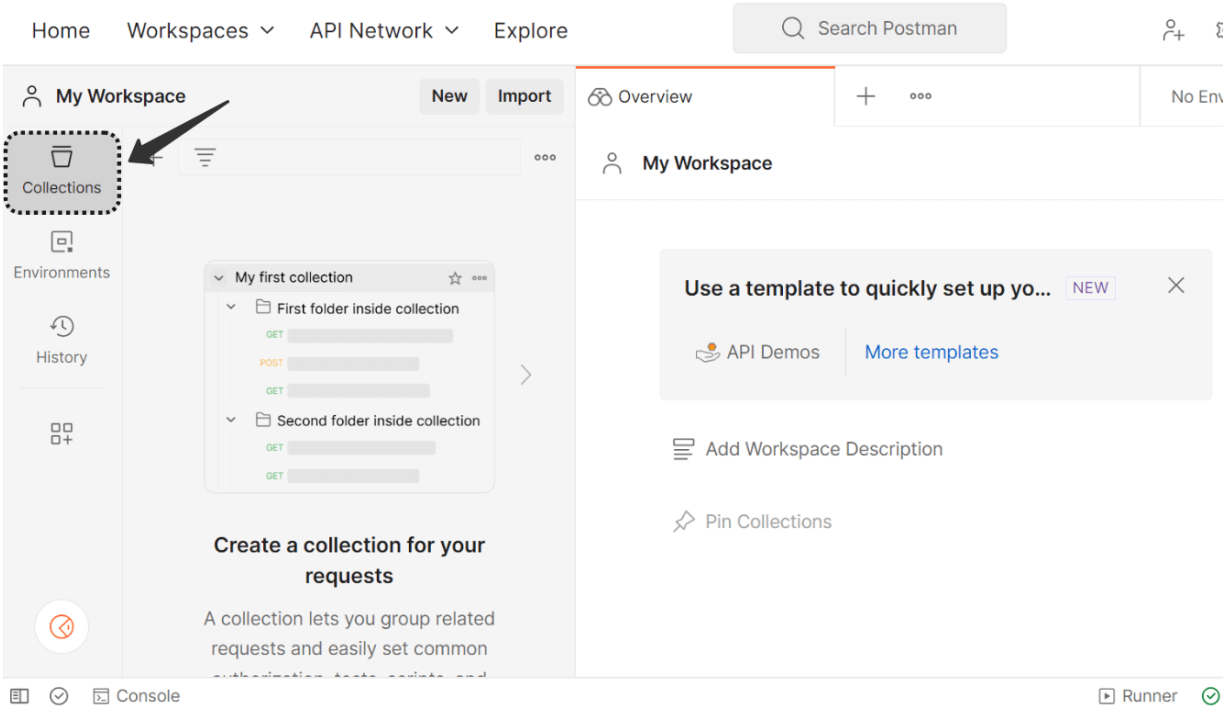


Figura 17.24: Workspace. Fonte: Postman.

Em *Collections* (coleções) fica a lista de projetos. Vamos criar uma coleção clicando no botão com sinal de "+".

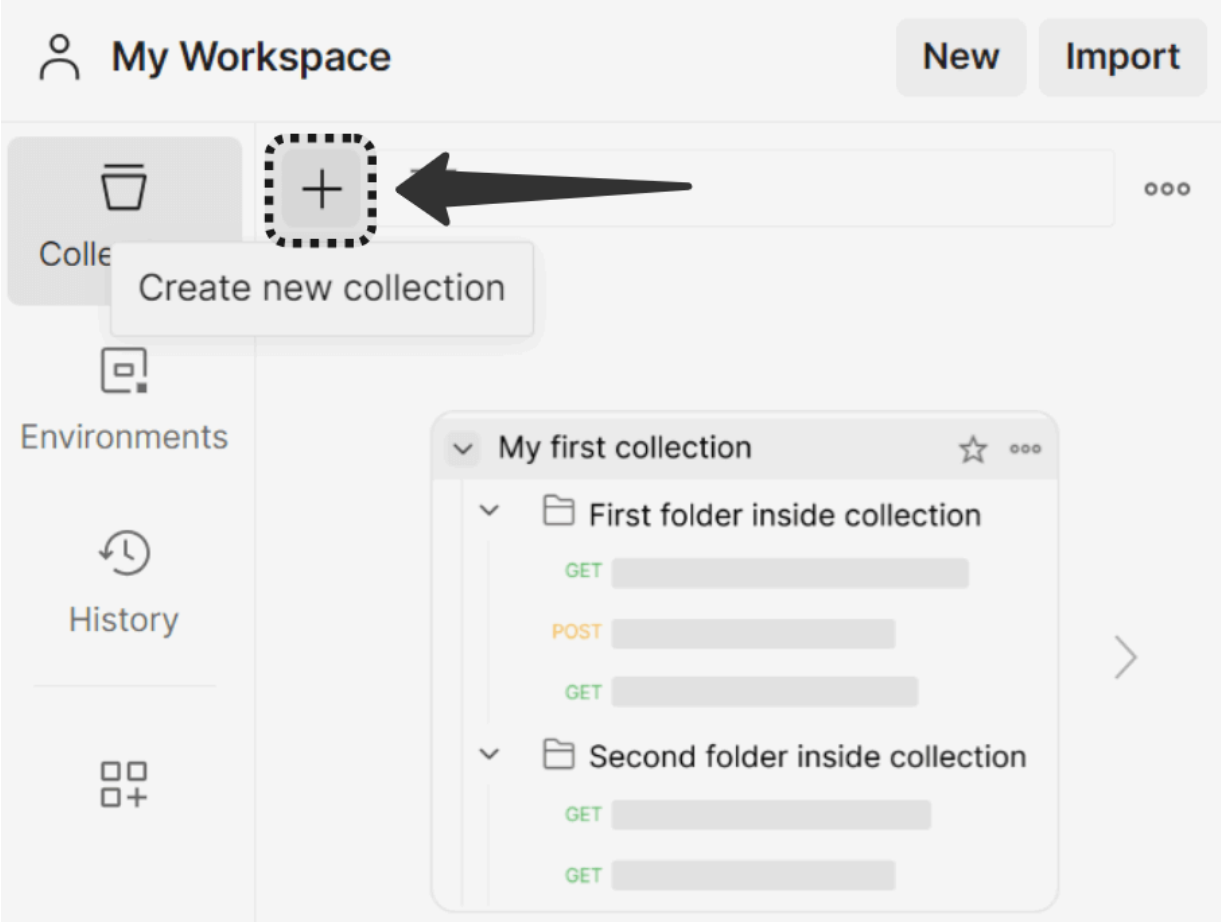


Figura 17.25: Fonte: Postman.

Isto feito, abrirá uma nova aba no lado direito chamada "*New Collection*", como apresenta a próxima imagem. Repare que o nome do projeto é exibido em diversos locais da página.

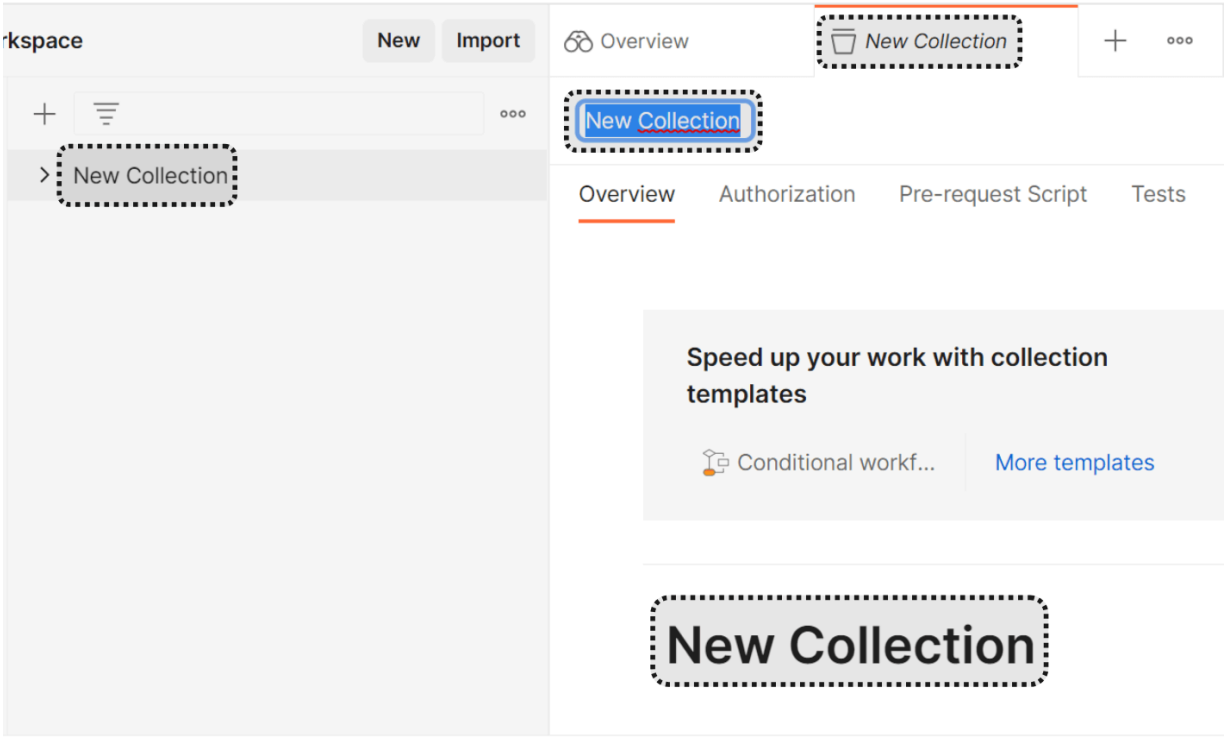


Figura 17.26: Criação de coleção. Fonte: Postman.

A API Places da Plataforma Google Maps exige API key e, para isso, é necessário fornecer um cartão de crédito. Então, vamos usar uma que é possível acessar sem a necessidade de API key, a Swagger Petstore. No Postman, sugiro colocar o mesmo nome — poderia ser qualquer outro, mas, dessa forma, você sempre saberá a qual API se refere.

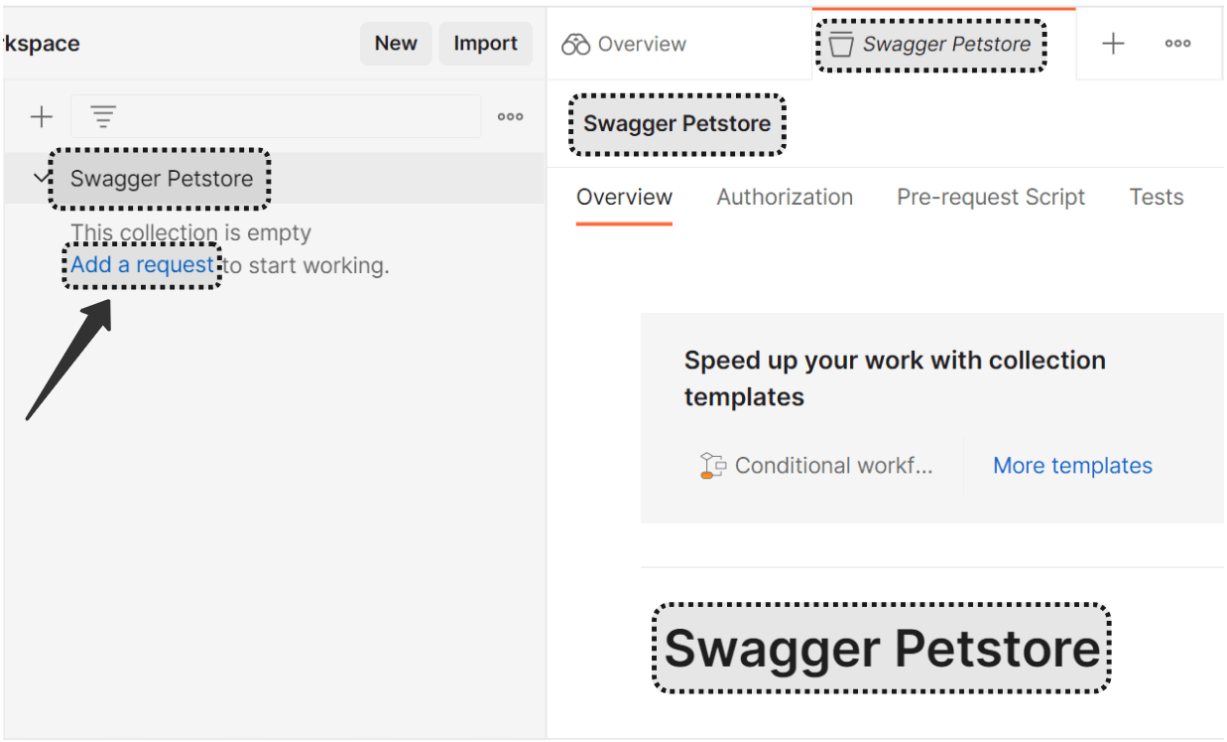


Figura 17.27: Criação de requisição. Fonte: Postman.

Ao clicar em "*Add a request*", apontado na figura anterior, abre mais uma aba com três áreas, dessa vez para configurar o teste:

1. A área com o menu *dropdown* para selecionar o método e o campo onde se insere a URL do *endpoint* da API;
2. A área "*Params*", com os campos para inserir os parâmetros — dados da consulta;
3. A área "*Response*", onde a resposta da API aparecerá.

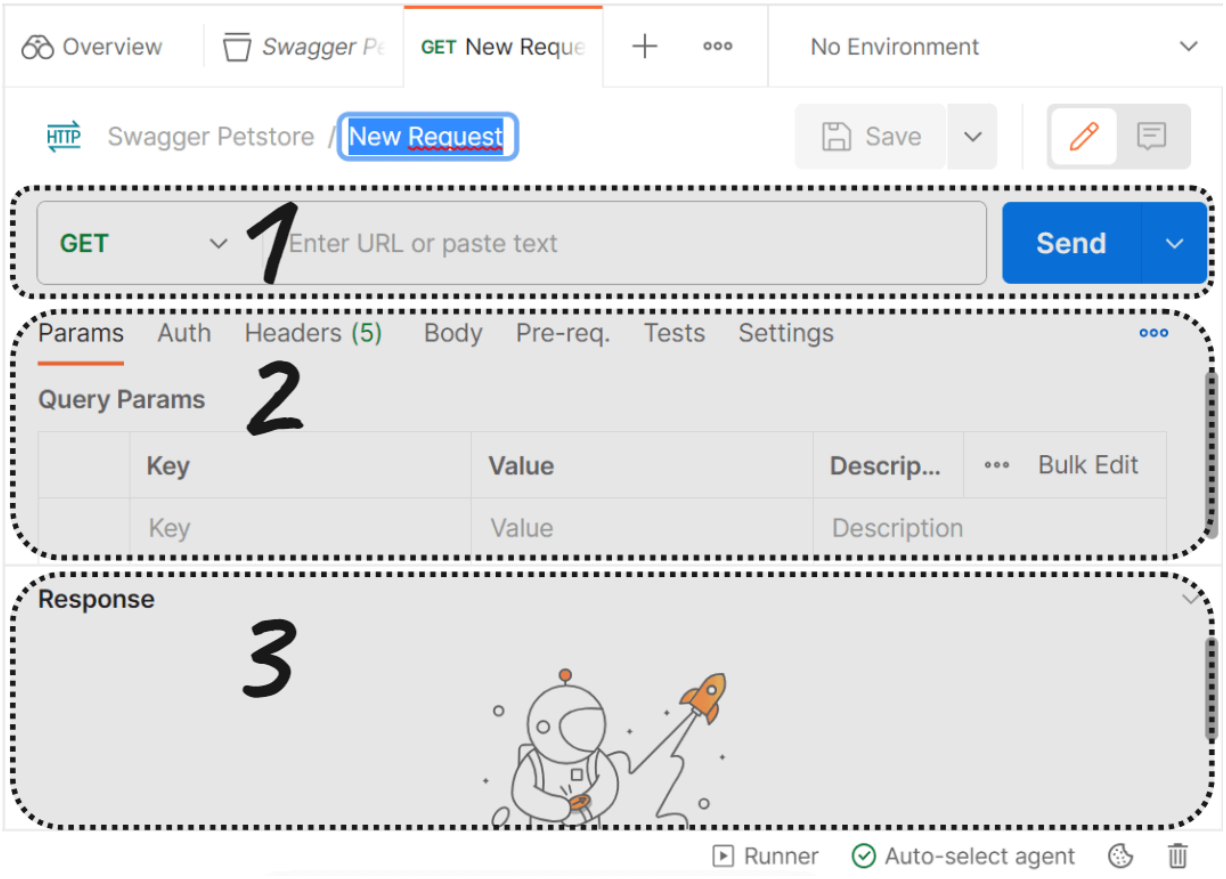


Figura 17.28: Configuração de requisição. Fonte: Postman.

Agora, é preciso acessar a documentação da API que será testada no Postman.

Acessando a documentação

Para acessar a API Swagger Petstore digite no navegador a seguinte URL: https://petstore3.swagger.io/. Você vai se deparar com a tela como da imagem adiante e repare que se trata da versão 3.0 — a versão faz diferença, porque o funcionamento pode ser consideravelmente diferente.

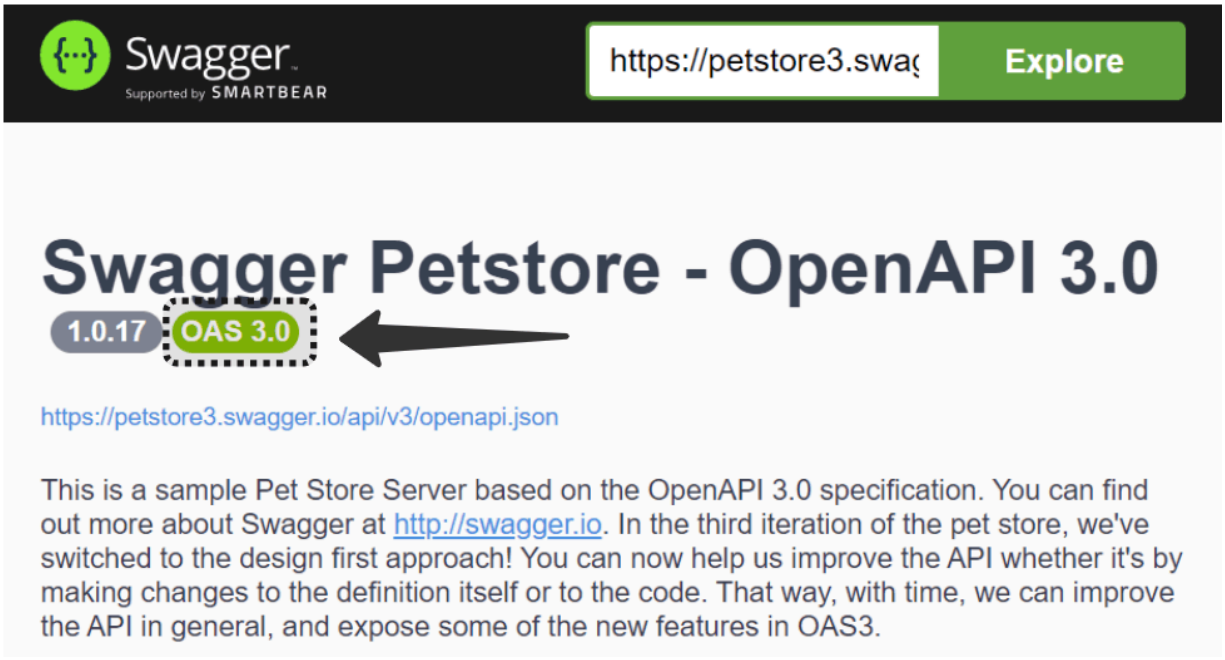


Figura 17.29: Fonte: Swagger Petstore v3.

O Swagger Editor, software em que a Swagger Petstore foi criada, possui ferramentas para design e consumo de APIs que seguem o padrão **OpenAPI** — uma especificação para APIs REST definida em JSON ou YAML. Junto da especificação, é gerada a documentação da API com os recursos, autenticação, métodos disponíveis, parâmetros, formatos de entrada e saída. Já a Swagger Petstore é uma API de demonstração, aberta e livre para uso. É um exemplo de como projetar e documentar APIs, muito usada por devs para experimentar e aprender a especificação OpenAPI.

A Swagger Petstore fornece três *endpoints* — *pet*, *store* e *user* — que permitem operações básicas de um pet shop, como gerenciar o cadastro de animais, pedidos de compra e contas de usuário. Cada *endpoint* possui um conjunto de métodos para criação, atualização, consulta e exclusão de registros. A figura adiante mostra os métodos disponíveis para pets:

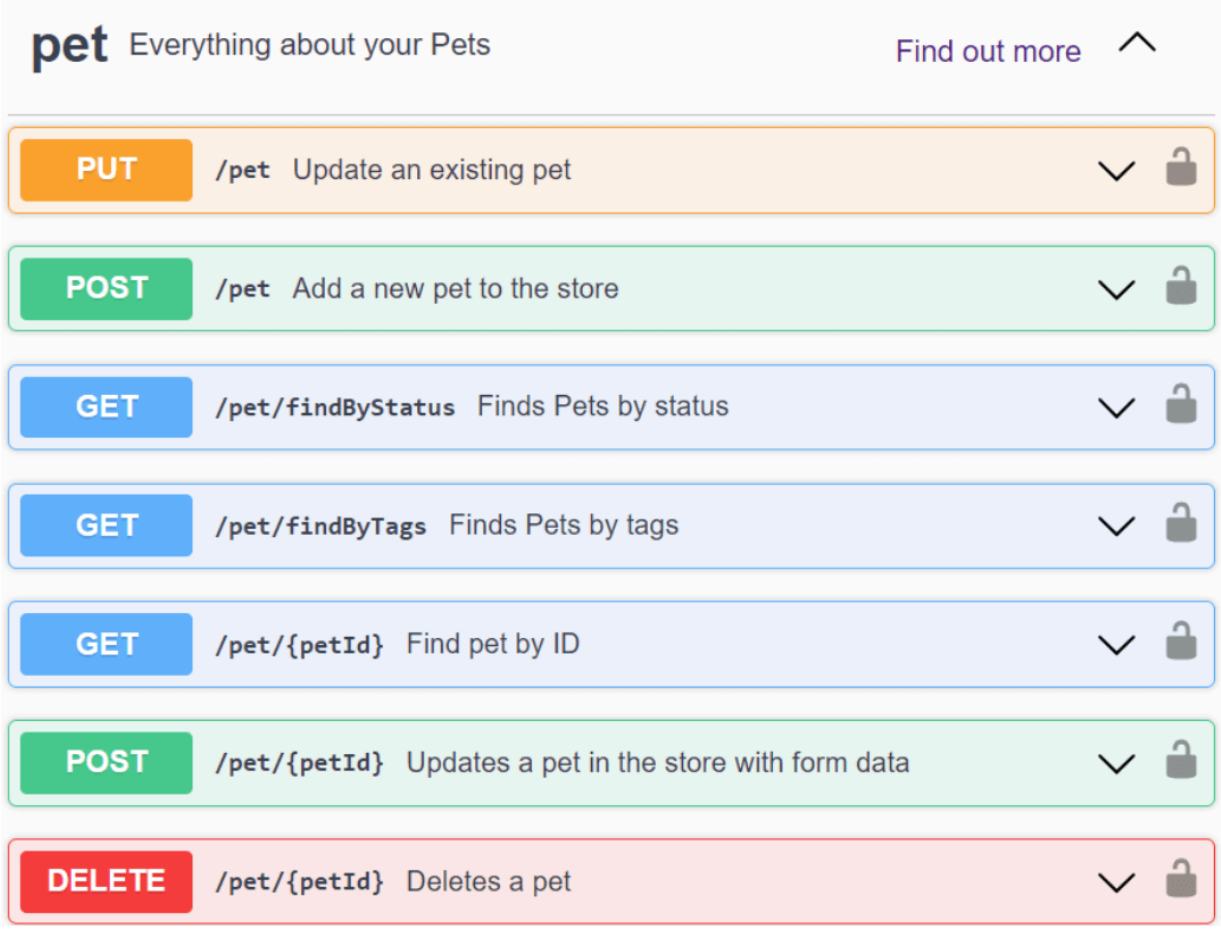


Figura 17.30: Métodos. Fonte: Swagger Petstore v3.

No final da documentação encontra-se os *schemas*, modelo ou modelagem dos dados que informam como os dados são organizados, formatos, relacionamentos, restrições e o tipo de dados permitidos, por exemplo, *integer* (numérico) ou *string* (texto).

Figura 17.31: Modelo de dados. Fonte: Swagger Petstore v3.

Ao longo do teste conheceremos a documentação em detalhes.

Listando cadastros

Primeiro, vamos buscar pets pelo status (*find pets by status*). Veja que, para este caso de uso, deve-se utilizar o método `GET` :

Figura 17.32: Filtro por status. Fonte: Swagger Petstore v3.

Ao clicar sobre a seta à direita, é exibida uma breve descrição informando que são aceitos múltiplos valores para status, é só separá-los por vírgula (*multiple status values can be provided with comma separated strings*). Mais abaixo, a seção "*Parameters*" descreve que o atributo `status` pode ser filtrado por *available*, *pending* e *sold* — disponível, pendente e vendido, respectivamente.

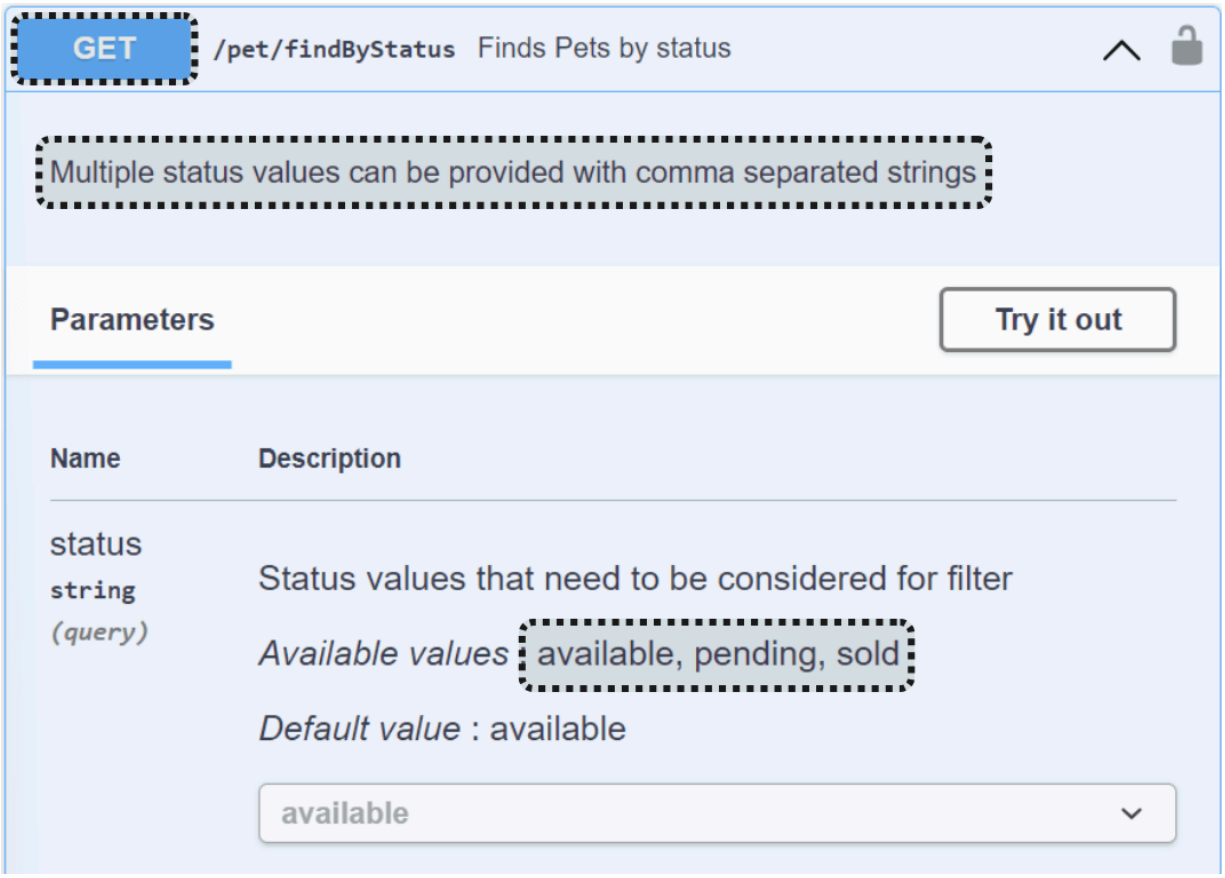


Figura 17.33: Requisição filtrada por status. Fonte: Swagger Petstore v3.

Sabendo que se trata de uma loja, conseguimos inferir o significado de disponível (para venda) e vendido, mas o que seria "pendente"? A documentação não diz. Então, quando usá-lo? Este é o caso típico em que os times usarão como quiserem, inclusive atribuirão significados sem qualquer relação com "pendente" se precisarem de um status que a API não fornece. Dentro da mesma empresa, esse cenário pode gerar problemas na análise e interpretação dos dados, por exemplo. Por isso, é importante definir o significado e

documentá-lo. Se surgir necessidade de outros status, o time da plataforma pode criá-los e, assim, manter a consistência dentro da organização.

A próxima figura mostra a seção "*Responses*", que apresenta o que esperar de resposta. Em caso de sucesso, código 200 e o exemplo de código de resposta em JSON; em caso de erro, código 400 e a exibição da mensagem "*Invalid status value*".

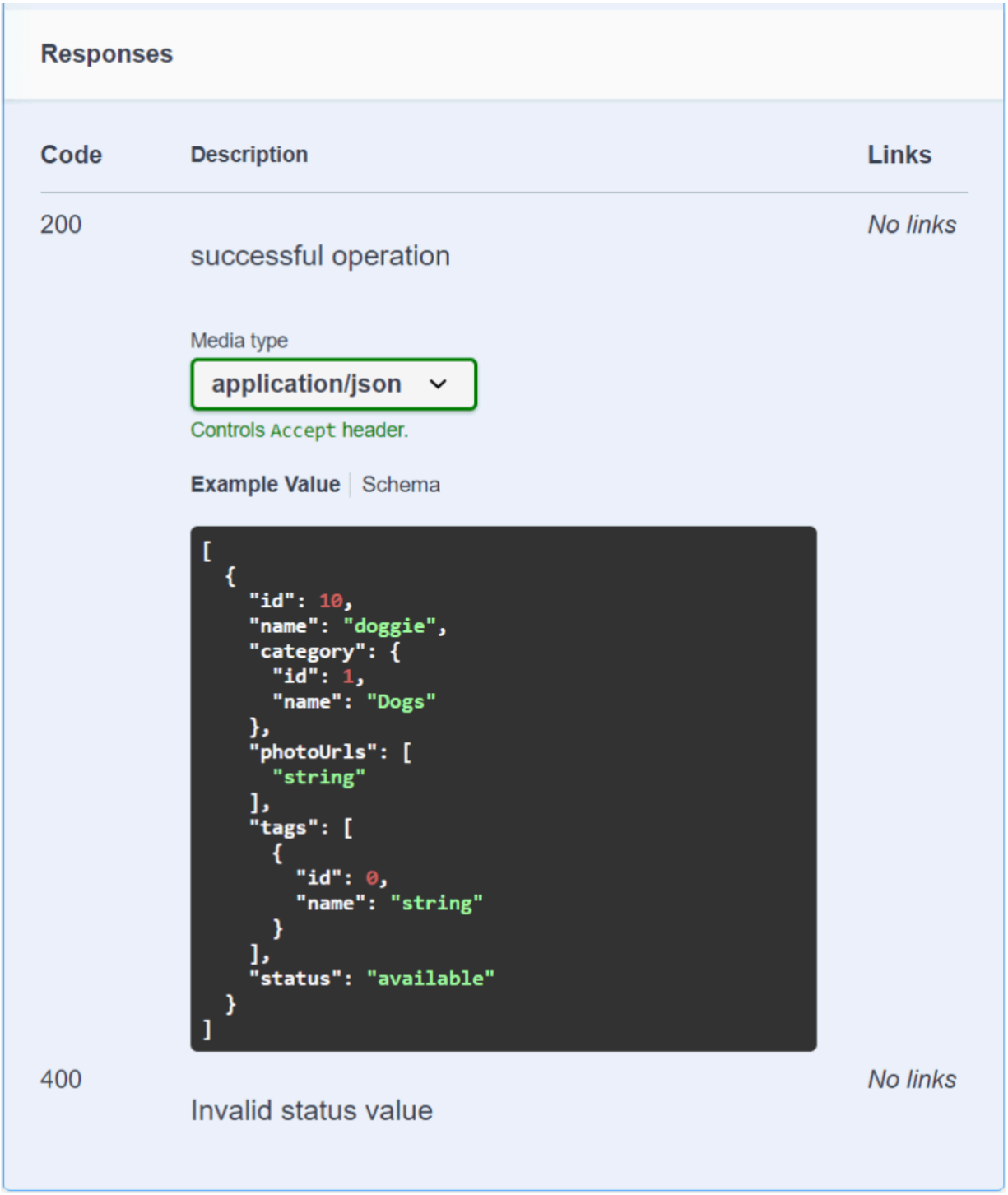


Figura 17.34: Resposta. Fonte: Swagger Petstore v3.

Você consegue fazer um teste simples na própria documentação ao clicar no botão "*Try it out*" na seção "*Parameters*". Veja na imagem adiante que, apesar de permitir a busca por mais de um status, não

é possível selecionar mais de um no *dropdown*. Para este cenário, podemos usar o Postman. Por enquanto, vejamos o que acontece quando queremos listar os pets disponíveis para venda. Para isso, vamos selecionar "*available*" no *dropdown* e clicar no botão "*Execute*".

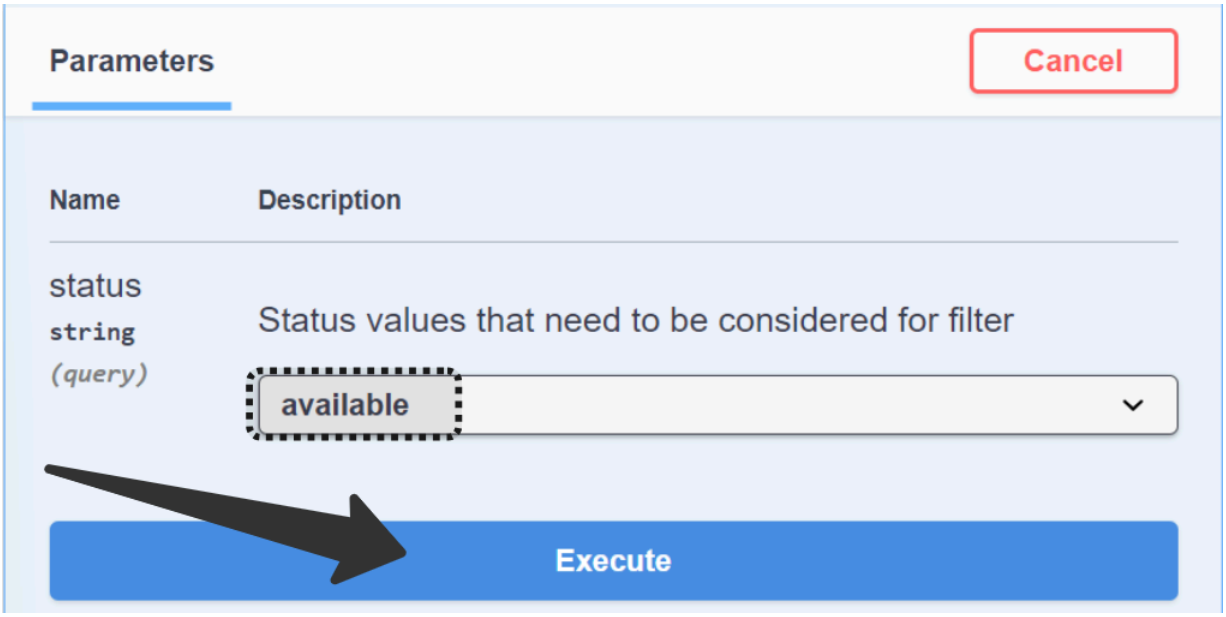


Figura 17.35: Filtro por status. Fonte: Swagger Petstore v3.

Isto feito, a ferramenta revela a resposta. No *print* seguinte podemos conferir a URL em que ocorreu a requisição, com o parâmetro `status=available` ; abaixo, o código 200 (sucesso) e o *payload* que foi retornado, a lista de pets cujo status é *available*. Note que os valores podem não fazer sentido. Por exemplo, `photoUrls` deveria conter uma URL, e o que vemos é um texto qualquer. Isso acontece porque as pessoas que testam fornecem os valores, e não há validação, então pode ser qualquer valor.

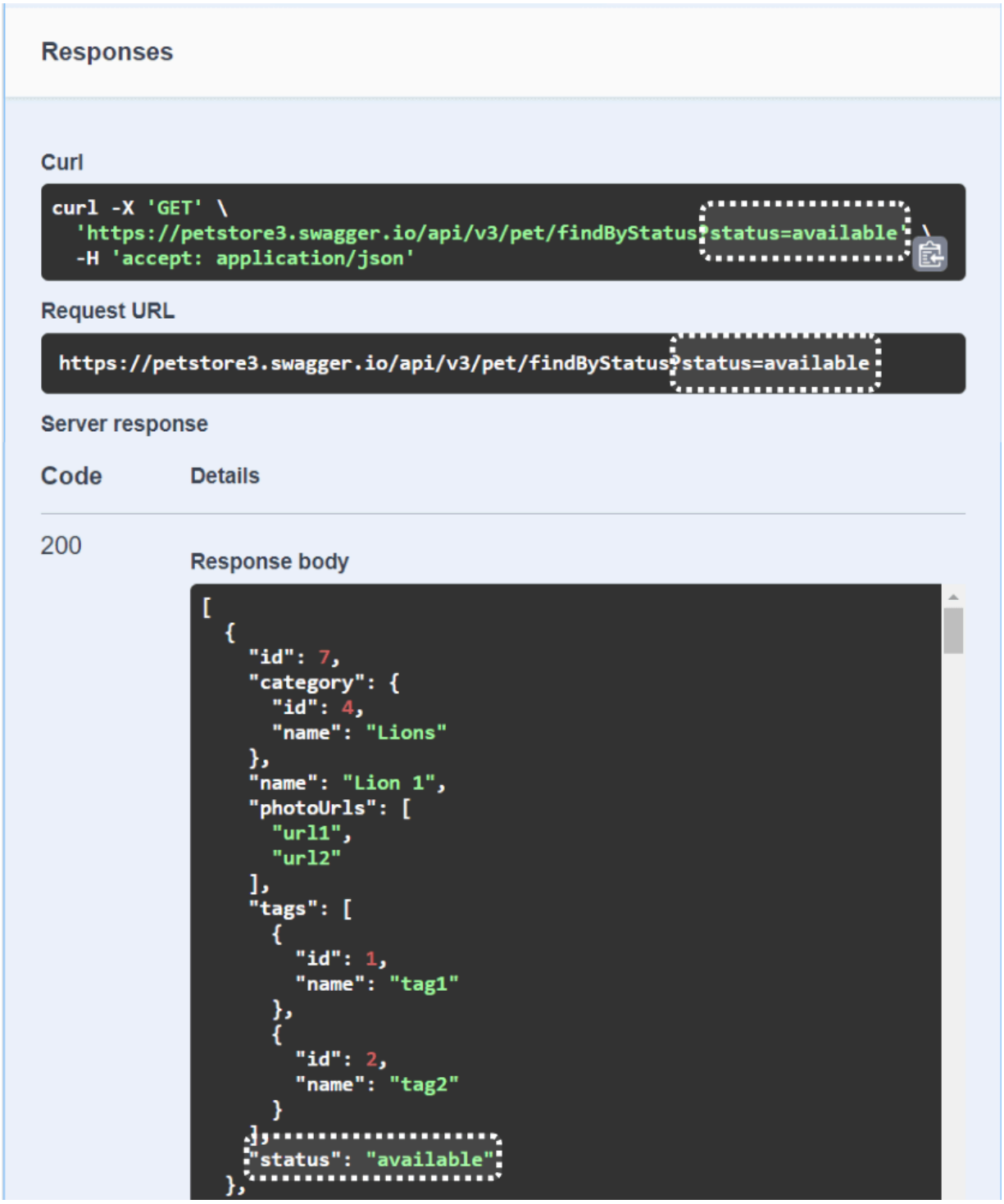


Figura 17.36: Teste da filtragem por status. Fonte: Swagger Petstore v3.

Para testar no Postman, copie a *Request URL* da documentação (`https://petstore3.swagger.io/api/v3/pet/findByStatus?status=available`)

e cole no campo com o *placeholder* "*Enter URL or paste text*". Lembra que a documentação aponta que se deve usar o método `GET` ? Assim, mantemos o método e em seguida é necessário clicar no botão "*Send*". Dessa forma, a requisição é enviada para o servidor da Swagger Petstore.

Figura 17.37: Requisição filtrada por status. Fonte: Postman.

Confira na imagem adiante que funcionou! Status "200 OK" e a resposta é a mesma que aparece no Swagger (`"id": 7`).

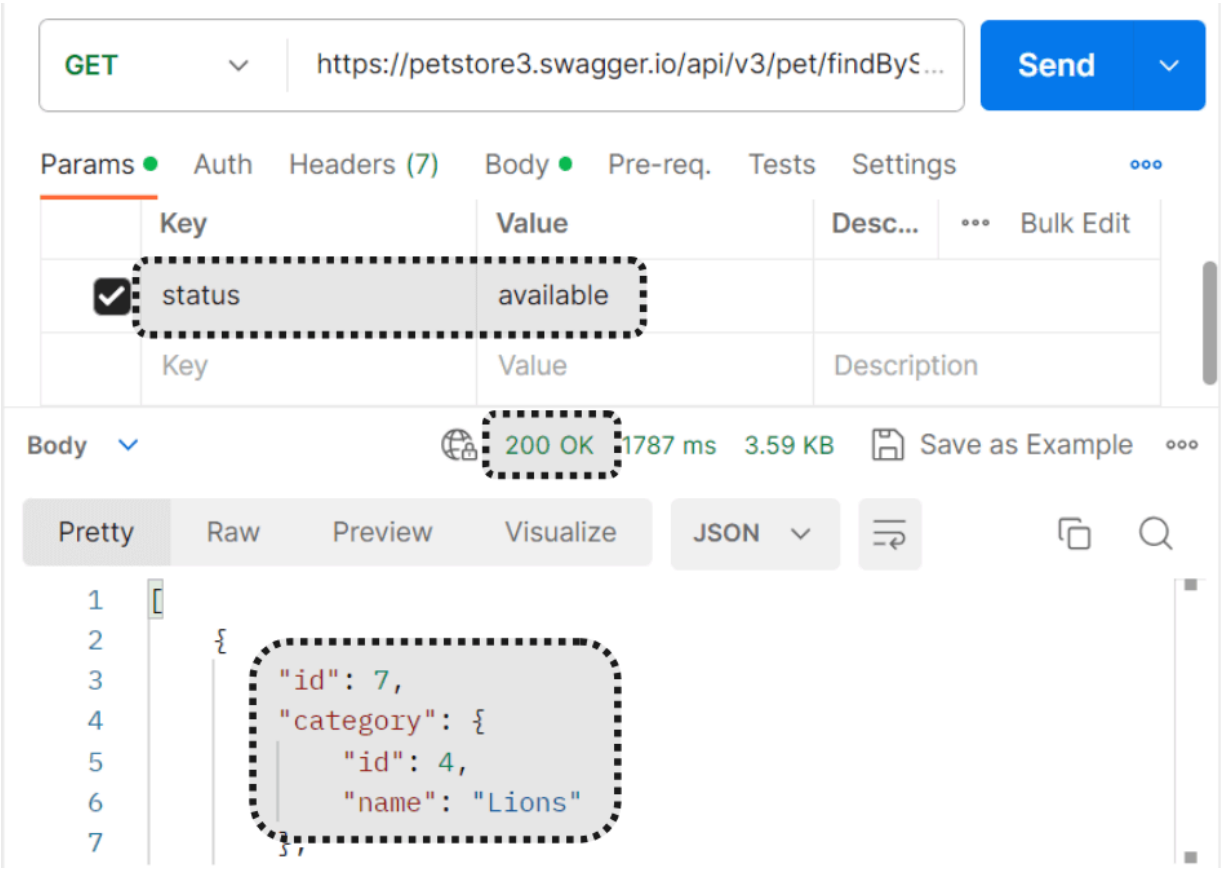


Figura 17.38: Resposta à requisição. Fonte: Postman.

Note que os campos em "*Params*" foram preenchidos automaticamente. Isso significa que você pode usar a interface visual para experimentar a API. Altere o "*Value*" do status para "*pending*" e olhe o que acontece:

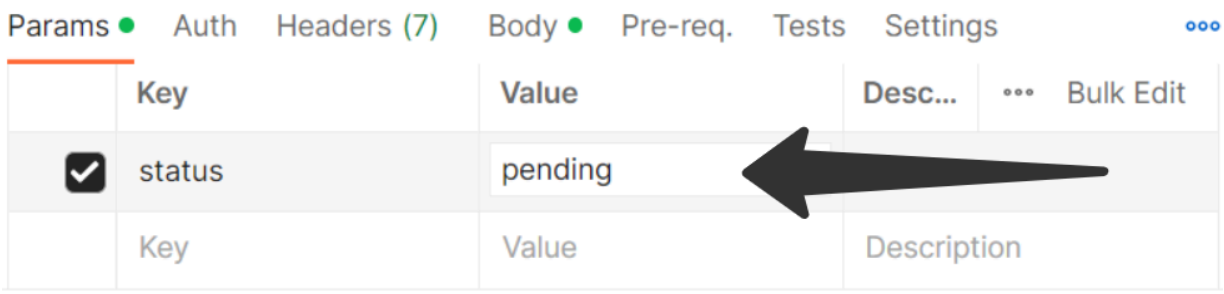


Figura 17.39: Filtro por status. Fonte: Postman.

O parâmetro na URL mudou automaticamente para "*pending*" também e que a resposta agora é diferente da anterior. O primeiro ID que apareceu para "*available*" era 7; para "*pending*", o primeiro resultado é o ID 1.

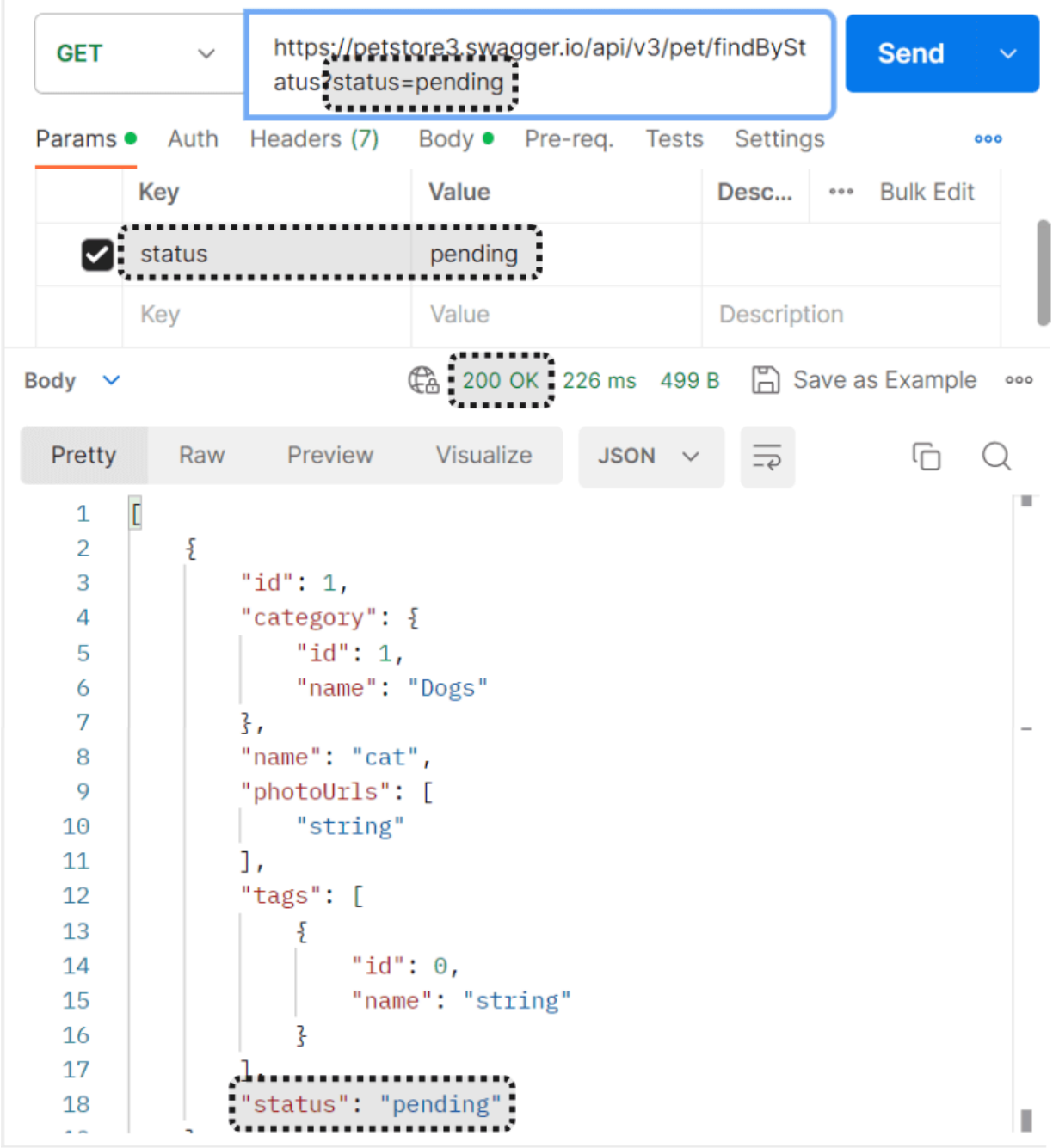


Figura 17.40: Resposta da filtragem por status. Fonte: Postman.

Vamos ver se realmente suporta múltiplos status? Segundo a documentação, cada valor deve ser separado por vírgula, como mostra a próxima figura:

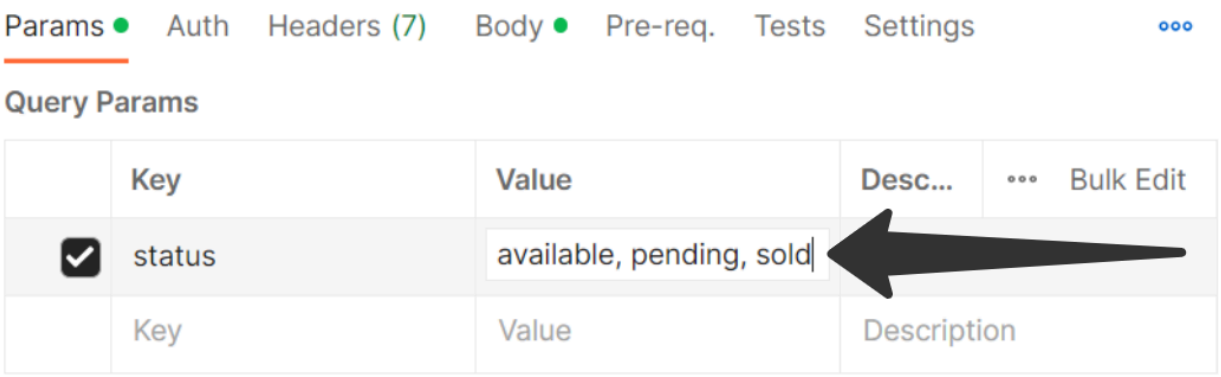


Figura 17.41: Filtro por mais de um status. Fonte: Postman.

400 Bad Request! No campo de resposta, a mensagem completa: *Input error: query parameter* `status value` *available, pending, sold is not in the allowable values* `[available, pending, sold]` . O sistema interpretou "*available, pending, sold*" como um único valor, não três.

Figura 17.42: Resposta da filtragem por mais de um status. Fonte: Postman.

Então, vamos separar os valores usando os campos em "*Params*": para cada linha, uma `Key` (chave) preenchida com "status" e, em `Value` , um único valor, como na figura a seguir.

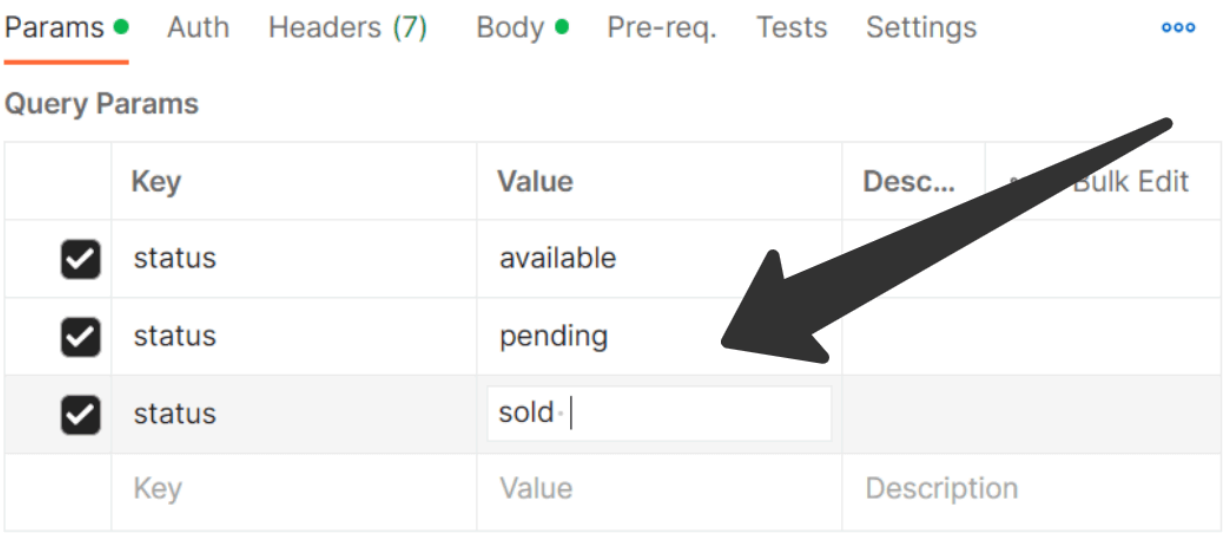


Figura 17.43: Filtro por mais de um status. Fonte: Postman.

Perceba que a URL mudou. Cada par de chave-valor foi separado por `&` e, dessa vez, o retorno de sucesso (200) e, abaixo, o *payload* da resposta.

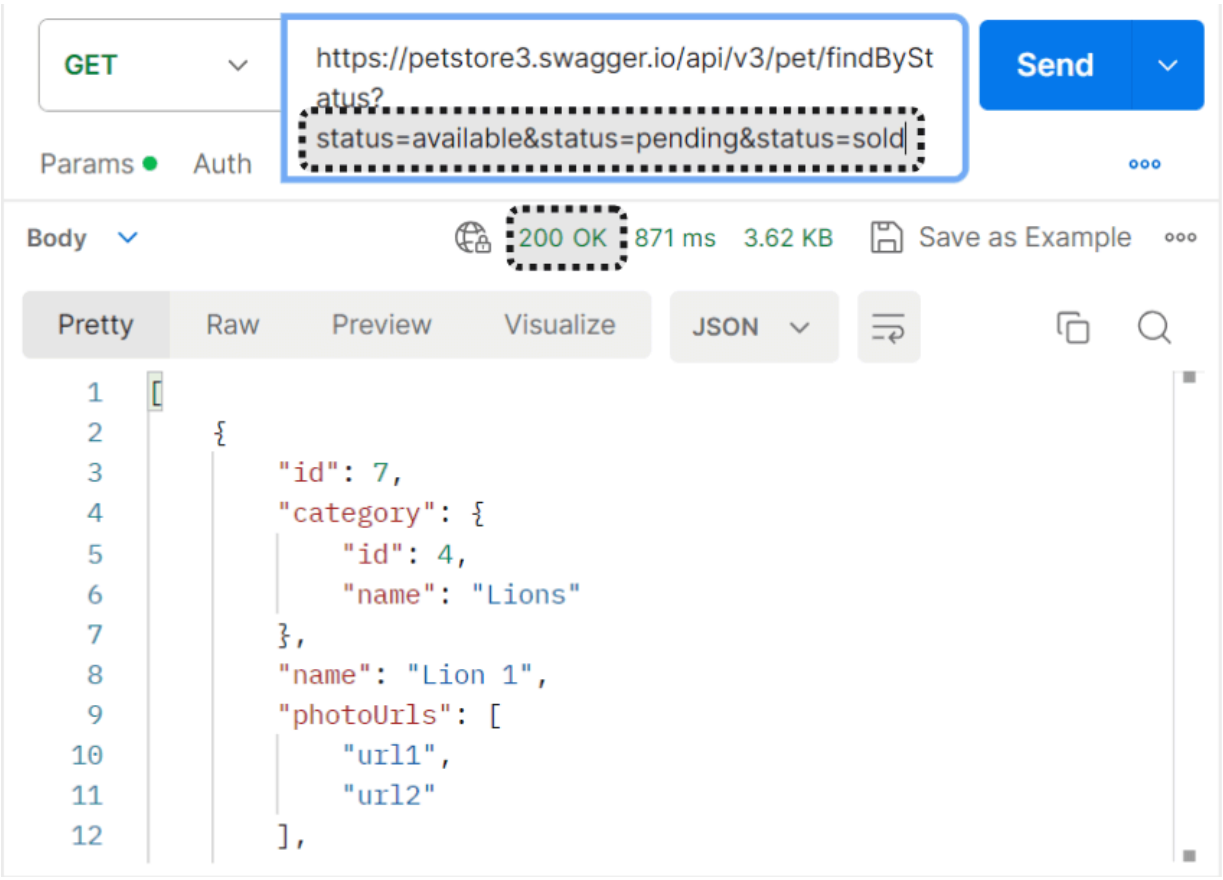


Figura 17.44: Resposta da filtragem por mais de um status. Fonte: Postman.

É fácil ou não? Será que é fácil assim para cadastrar um pet?

Criando um cadastro

De acordo com a documentação, para cadastrar um pet na loja ("*Add a new pet to the store*") deve-se usar o método `POST`, e é obrigatório enviar os dados no *Request body* (corpo da requisição). Ou seja, para criar um registro, é preciso passar os dados a serem registrados, do contrário, poderia ser criado um novo pet sem nenhuma informação. A documentação fornece um exemplo do que enviar e como estruturar a requisição.

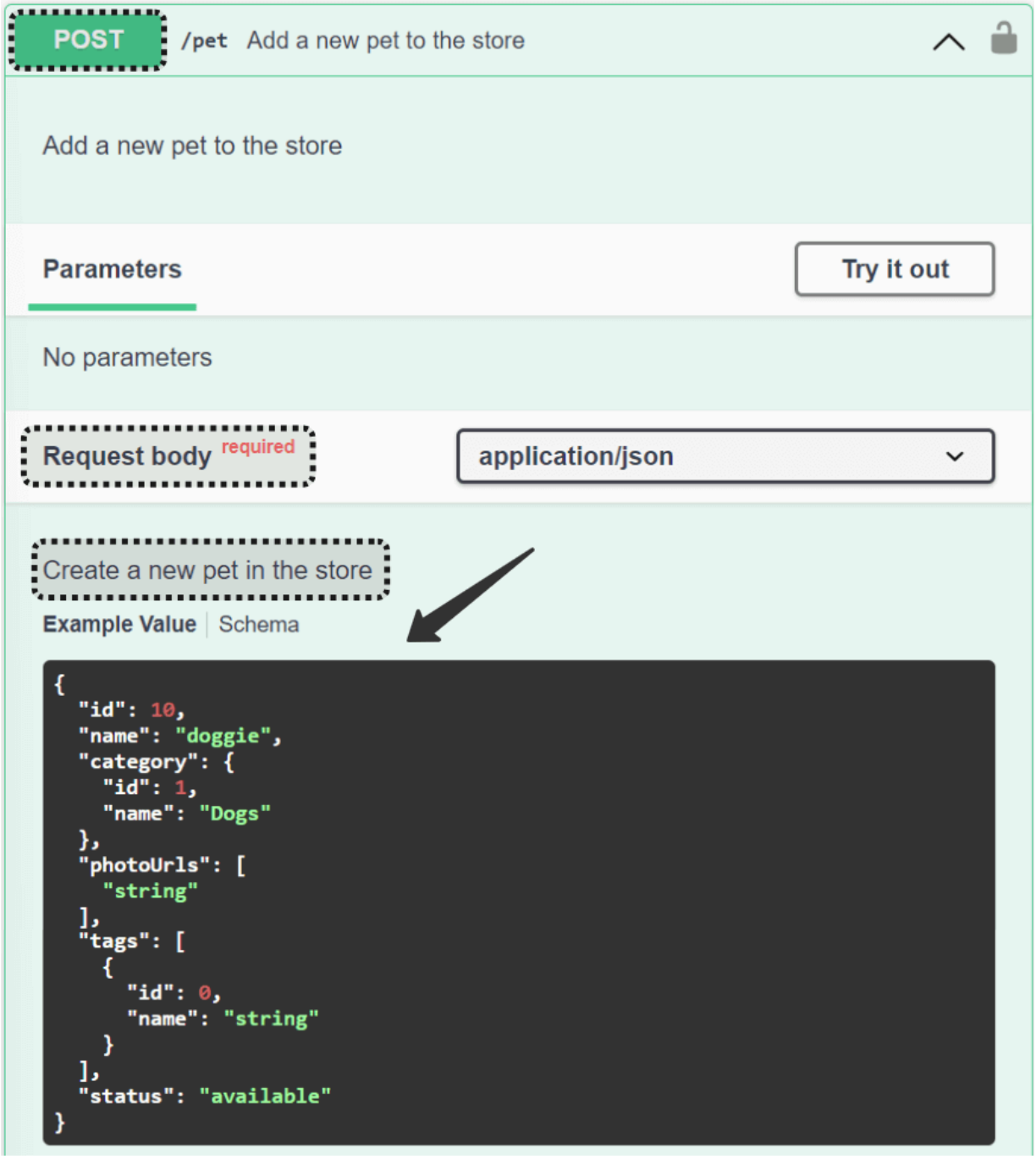


Figura 17.45: Criação de novo cadastro. Fonte: Swagger Petstore.

A próxima figura apresenta o exemplo de resposta, que é exatamente o que foi enviado na requisição:

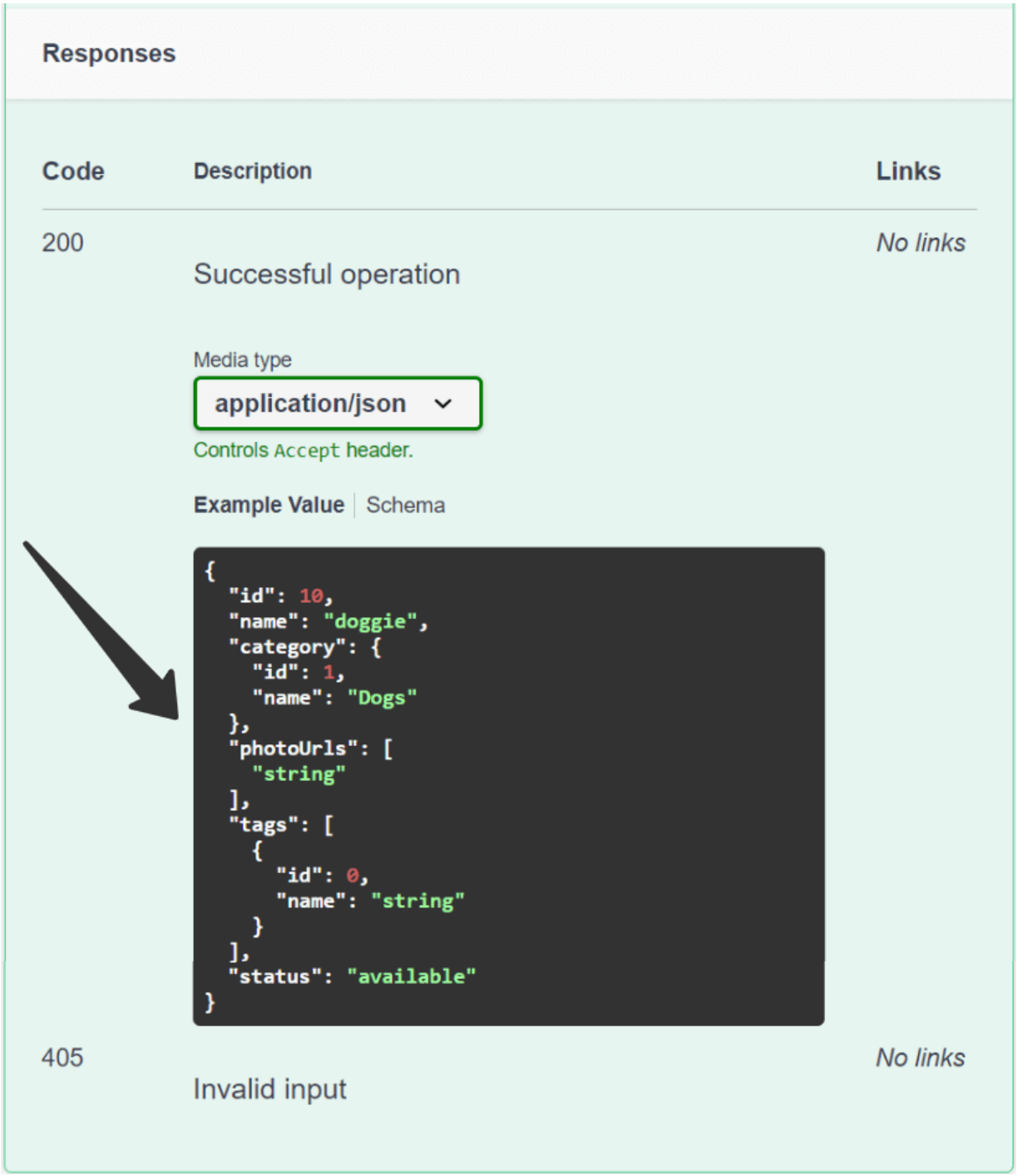


Figura 17.46: Exemplo de resposta. Fonte: Swagger Petstore

Vamos substituir os valores do exemplo para criar um cadastro de pet.

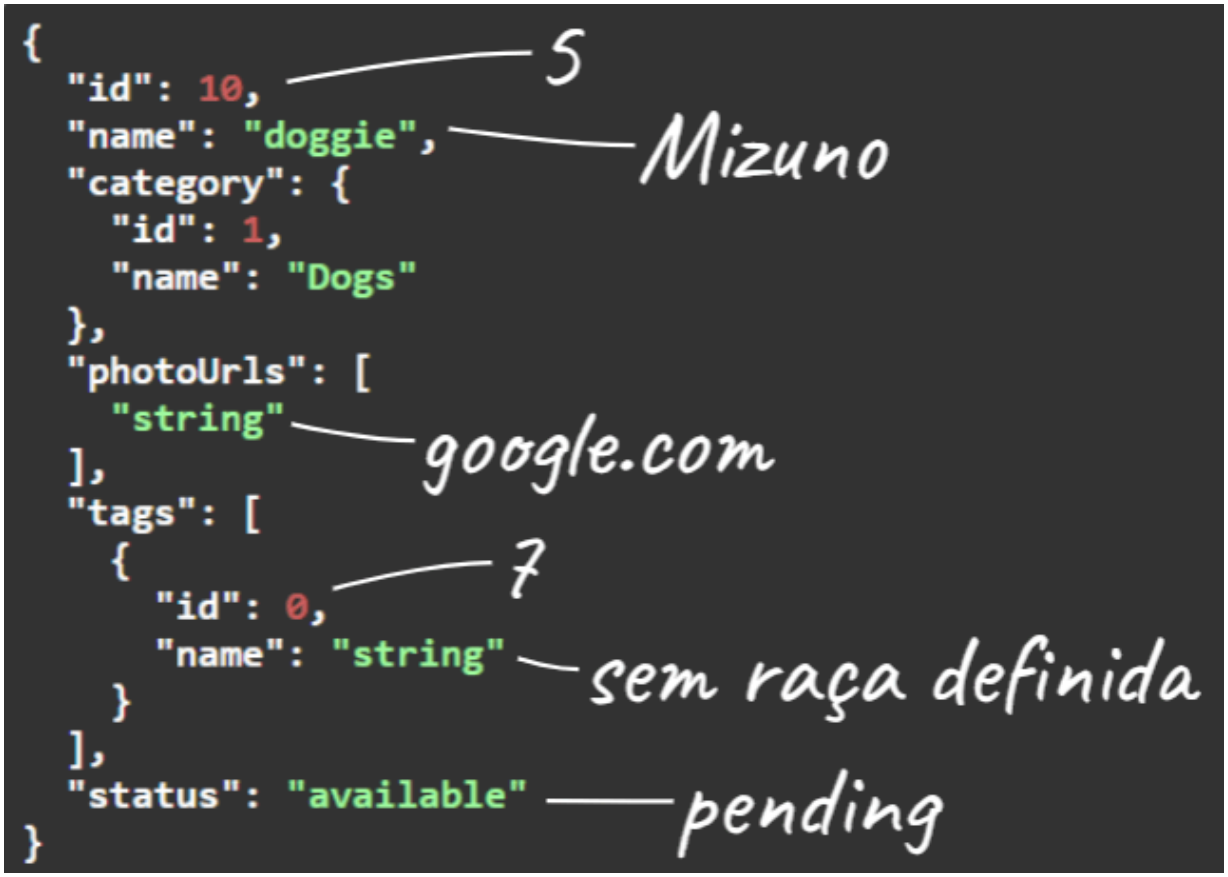


Figura 17.47: Dados do novo cadastro. Fonte: Swagger Petstore.

Vamos copiar o exemplo de código que está no Swagger e cadastrar o pet via Postman:

1. Primeiro, mude o método para `POST`;
2. Depois, insira a URL `https://petstore3.swagger.io/api/v3/pet`;
3. No menu, abaixo da URL, clique em "*Body*";
4. Abaixo do "*Body*", selecione "*raw*";
5. Mais para a direita, selecione a linguagem "JSON";
6. Cole no campo de texto o exemplo de código copiado no Swagger e altere os dados para os do pet a ser cadastrado — os dados "*category*" foram mantidos;
7. Clique em "*Send*" para executar a requisição.

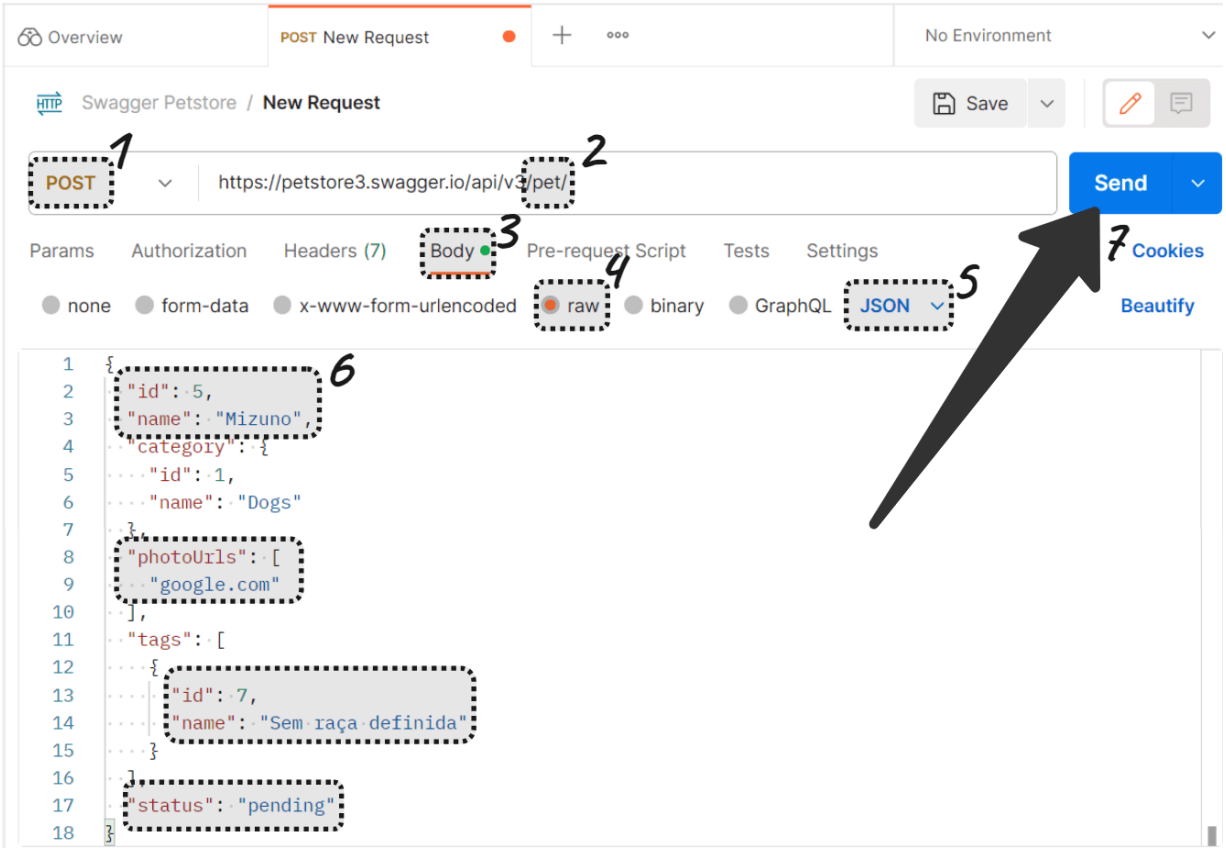


Figura 17.48: Requisição de novo cadastro. Fonte: Postman.

A resposta apresenta o código 200, o *payload* traz o ID 5 e os demais valores conferem. Resumindo, foi cadastrado com sucesso. Será que foi mesmo?

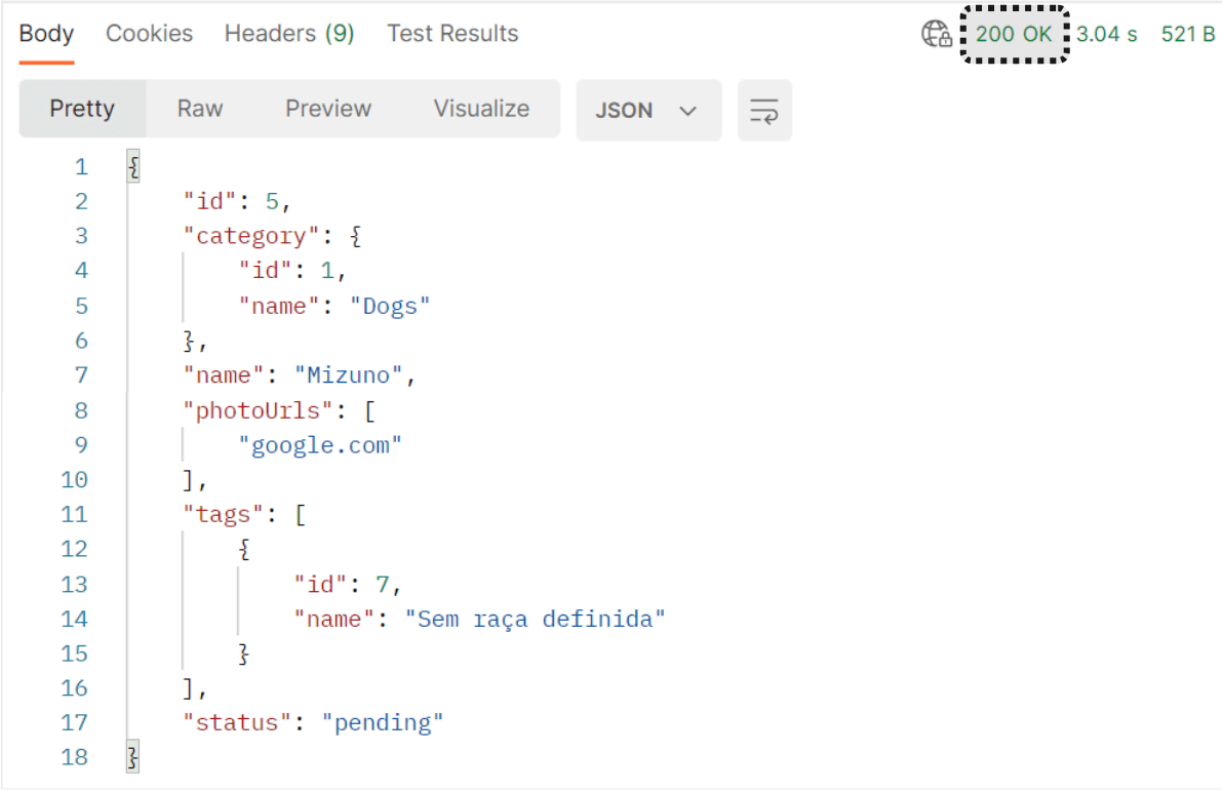


Figura 17.49: Resposta. Fonte: Postman.

Buscando um cadastro

Segundo a documentação, para buscar um pet específico deve-se usar o método `GET` e na URL, depois de `/pet/`, o respectivo ID do pet, representado por `{petId}`. A seção "*Parameters*" aponta que o `petId` é um parâmetro obrigatório na requisição — e não poderia ser diferente, afinal, sem essa informação, como o sistema saberá qual pet retornar?

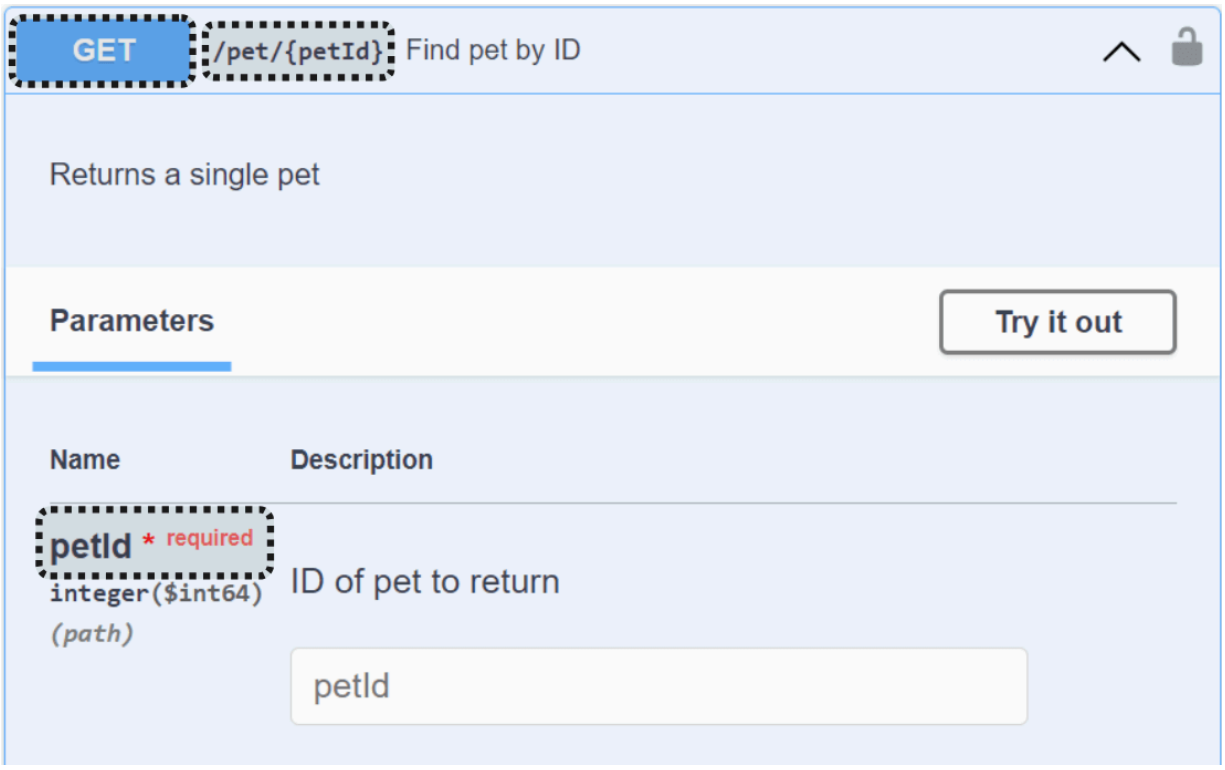


Figura 17.50: Busca de cadastro específico. Fonte: Swagger Petstore.

O ID do pet cadastrado no teste via Postman é 5. Então, vamos mudar o método para `GET` e inserir o número 5 na URL — `https://petstore3.swagger.io/api/v3/pet/5`.

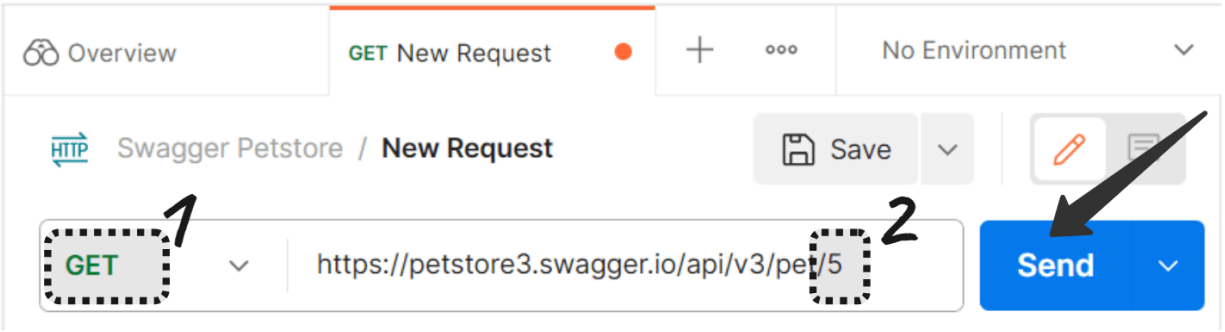


Figura 17.51: Busca pelo ID. Fonte: Postman.

Depois de clicar em "*Send*", todos os dados do pet devem ser retornados como resposta, de acordo com a documentação. Na próxima imagem, vemos os mesmos dados do pet cadastrado anteriormente — isto significa que foi criado mesmo!

Figura 17.52: Resposta. Fonte: Postman

Que tal confirmar? Na documentação, clicando no botão "*Try it out*" você ativa o campo para inserir o ID. Ao digitar 5 e clicar no botão "*Execute*", a resposta do servidor é exibida (*Server response*): ID *5*, nome *Mizuno* e status *pending*.

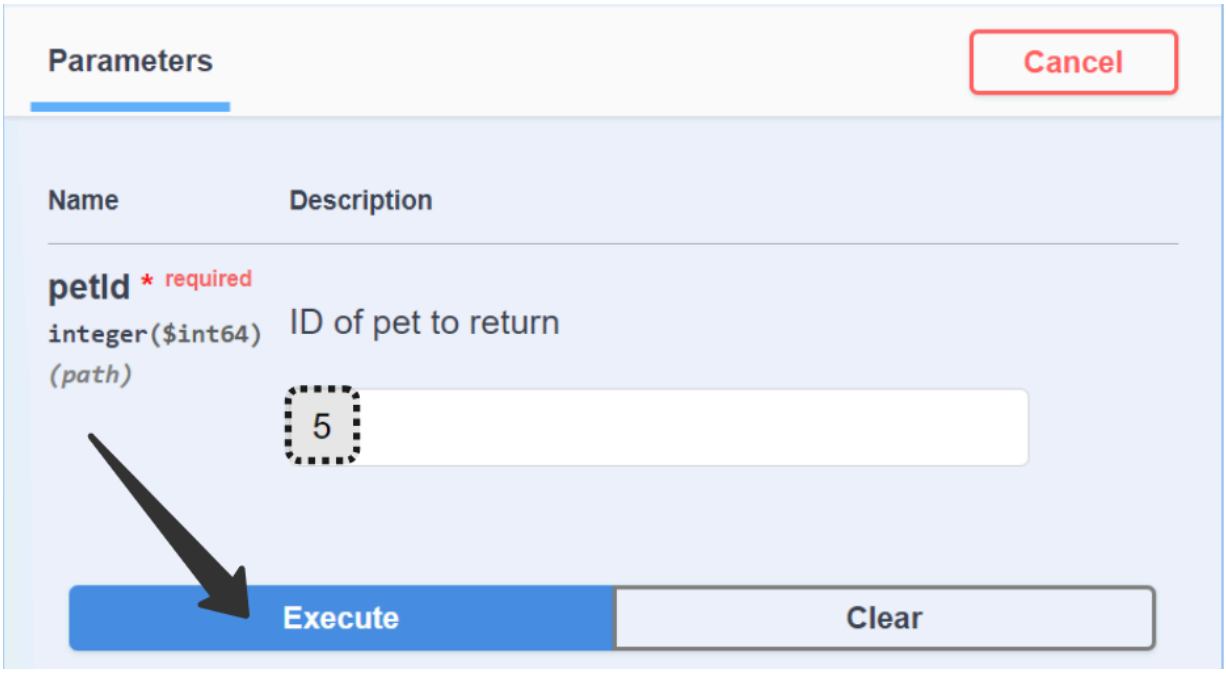


Figura 17.53: Busca pelo ID. Fonte: Swagger Petstore.

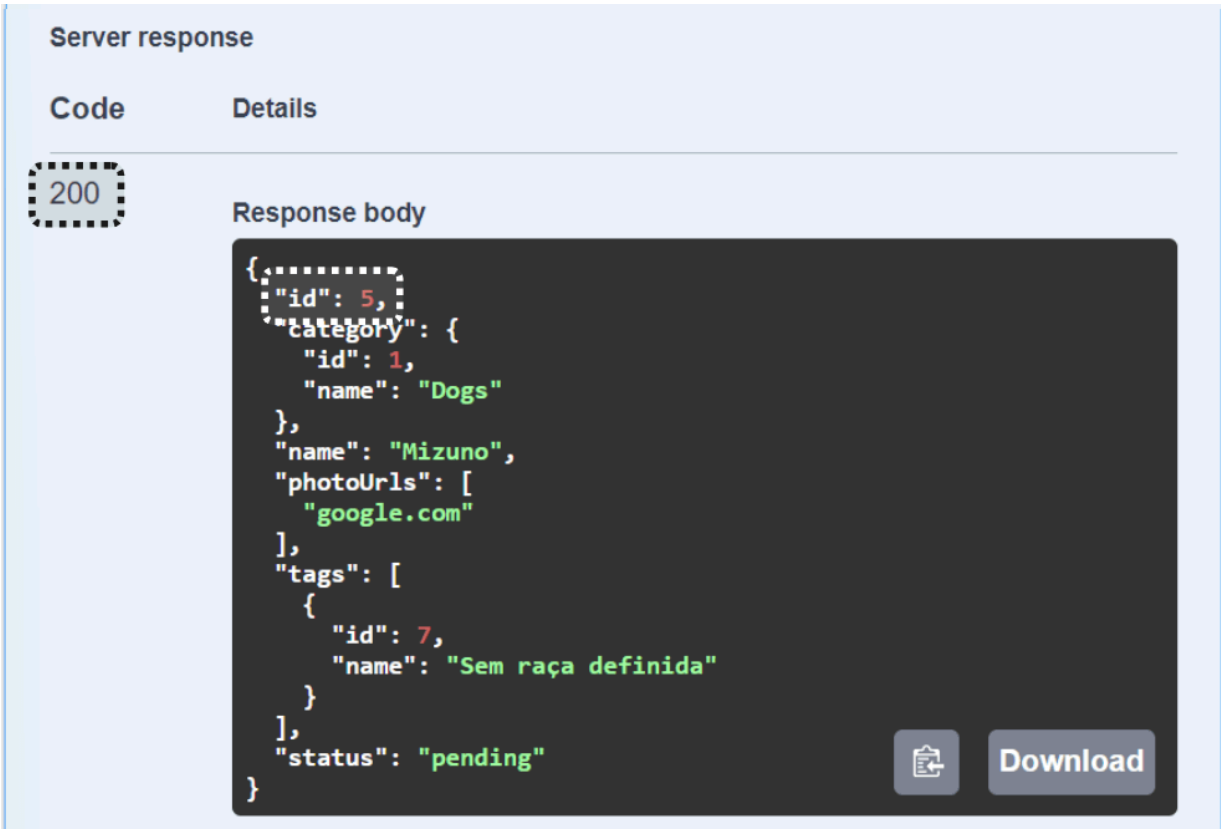


Figura 17.54: Resposta. Fonte: Swagger Petstore.

E se quisermos deletar esse cadastro?

Deletando um cadastro

A documentação, diz que para deletar um cadastro, usa-se o método `DELETE` e o ID do pet a ser descadastrado:

Figura 17.55: Exclusão de cadastro específico. Fonte: Swagger Petstore

No Postman, vamos inserir o ID 5 na URL para descadastrar o pet que criamos e mudar o método para `DELETE`. De resposta devemos receber o status 200 e no campo de texto "*Pet deleted*" (pet deletado), como a imagem.

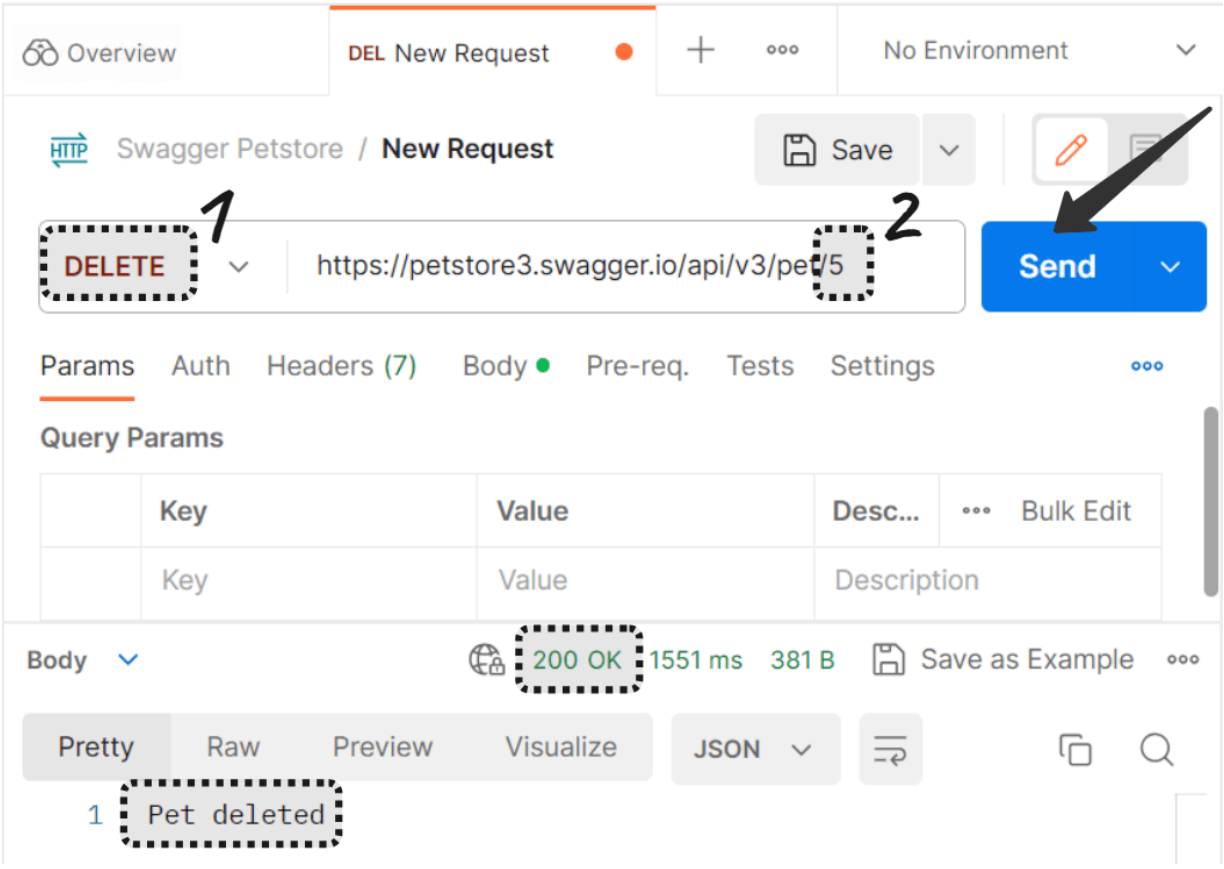


Figura 17.56: Exclusão de cadastro por ID. Fonte: Postman.

Quer saber se deletou mesmo? Basta mudar o método para `GET` e clicar no botão "*Send*" — o ID do pet já está na URL. Você obterá a como resposta "*Pet not found*". Note que o status mostra erro *404 Not Found*, isso quer dizer que o registro não foi encontrado. Esse é justamente o comportamento esperado, já que o pet foi descadastrado.

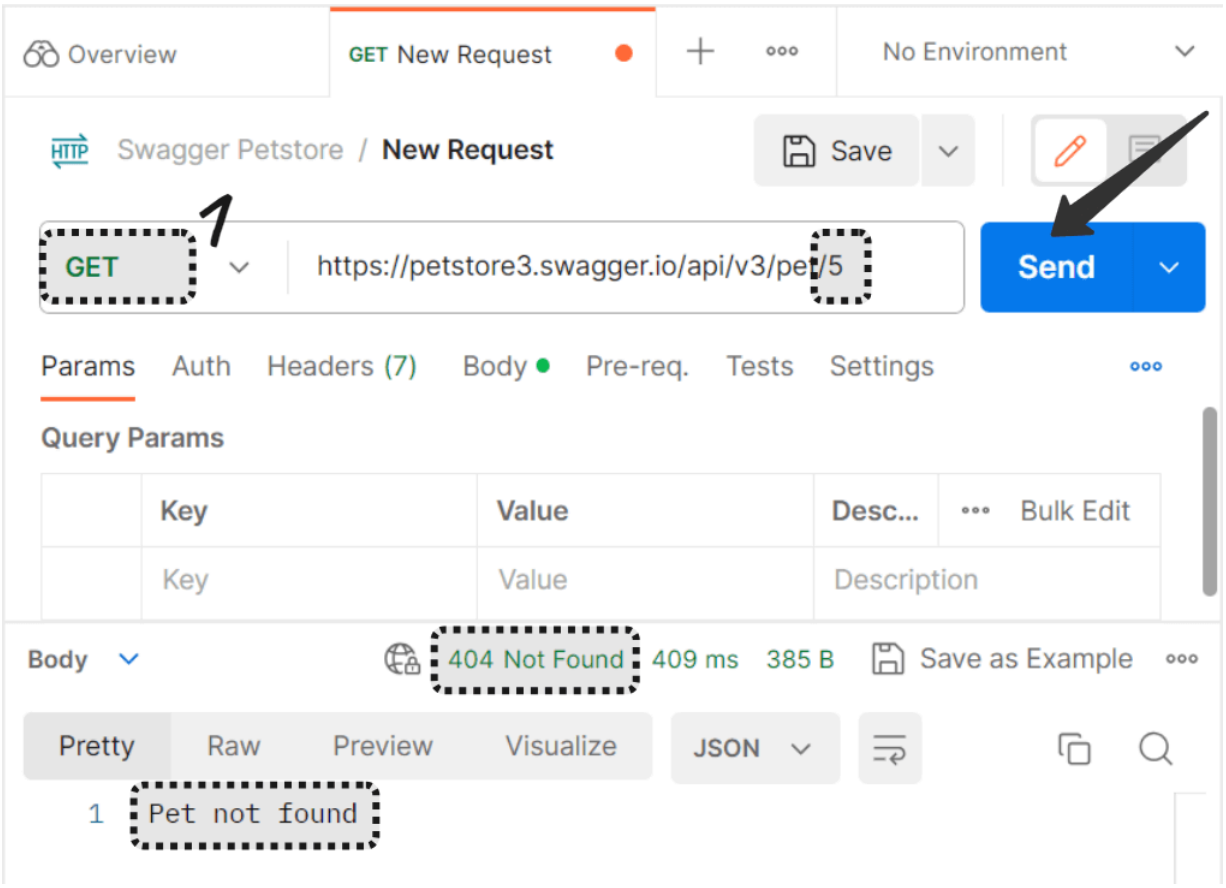


Figura 17.57: Busca de cadastro específico. Fonte: Postman

Agora você sabe como experimentar e testar APIs. Você percebeu que, sem a *documentação*, isso seria impossível? No próximo capítulo, você vai aprender como criar APIs, desde a ideação até o teste de usabilidade, e isso inclui a documentação.

Capítulo 18
Da ideação ao teste de usabilidade

Tratar uma API (ou plataforma) como produto tem por objetivo facilitar a adoção, acelerar o *time-to-market* e minimizar a necessidade de suporte do time que a desenvolve. Isso significa que o ponto de atenção central é torná-la ideal para consumo e isso não envolve apenas aspectos técnicos (engenharia), isso inclui as pessoas que atuam na integração, os devs usuários. É aqui que entra a experiência deles e a materialização disso inicia na fase de *Ideação* do Design Thinking.

Se o time não tem product designer ou API designer, é provável que você, PM, será a voz dos times clientes e terá o vislumbre de proporcionar a melhor experiência possível aos usuários da sua API.

Se esse é o seu caso, antes de partir para a ideação, é importante que você tenha noções do que avaliar em relação a *developer experience* e *usabilidade da API* — obviamente, sem a pretensão de dominar o assunto tanto quanto uma pessoa API designer.

18.1 Developer experience — DX

Se devs são as pessoas que usam a API, é a experiência delas que deve ser o foco de platform product managers na criação de uma API intuitiva e fácil de usar. Assim, os times dos apps conseguem oferecer valor e a experiência ideal aos usuários finais, dentro do *time-to-market* esperado pela empresa.

Developer experience (DX ou DevX) está para plataformas assim como **user experience (UX)** está para produtos e, da

mesma forma que a experiência do usuário final engloba todos os pontos de contato com o produto e a empresa, a experiência do dev contempla todas as interações que ele tem com a plataforma. Isso ocorre por meio do design da API e da documentação.

O que determina se uma API tem o design intuitivo e fácil de usar? A noção de aparência se dá pelas características e comportamento dos componentes da API, determinados pela sintaxe e semântica.

Sintaxe e semântica no design de API

Programação em geral é baseada em texto, lógica e cálculo. APIs não processam informação, só a trafegam, e isso se resume em texto, o que significa dizer que o *look and feel* de uma API REST é moldado pela estrutura e nomenclaturas dos componentes.

A **sintaxe** organiza a estrutura das requisições e das respostas da API — a relação lógica entre combinações possíveis de como as coisas são escritas para transmitir a semântica e, assim, serem compreendidas.

A **semântica** carrega o significado por trás dos nomes dos dados e das operações da API — o que as coisas representam e o sentido das relações. Vejamos como esses conceitos se aplicam na prática em alguns componentes de APIs REST.

Método

A sintaxe do método refere-se ao uso dos verbos HTTP para definir a ação a ser realizada sobre um recurso — `GET`, `POST`, `PUT`, `DELETE`. Note que o verbo é expresso em caixa alta. A semântica por trás desses verbos determina a operação a ser executada: `GET` para obter dados, `POST` para criar, `PUT` para atualizar e `DELETE` para excluir.

Endpoint

A sintaxe dos *endpoints* emprega substantivos no plural para indicar coleções de objetos — `/usuarios` —, e substantivos seguidos de um identificador para recursos específicos — `/usuarios/{id}` . A estrutura lógica das URLs é hierárquica, refletindo os relacionamentos entre os recursos, como em `/usuarios/{id}/pedidos` . Não se usa verbo nos nomes dos recursos, por exemplo `/criar-pedido` , pois o método HTTP já define o verbo. A semântica dos nomes dos *endpoints* deve refletir os recursos do domínio e funções, como `/usuarios` para informações de usuários, `/usuarios/{id}` para detalhes de um usuário específico, `/pedidos` para pedidos. Assim, `/usuarios/{id}/pedidos` indica os pedidos de um usuário específico.

Parâmetros

A sintaxe para parâmetros envolve a estrutura composta por pares de chave-valor adicionados após "?" e separado por "&", como em `/usuarios?nome=Pernalonga&idade=89` . Semanticamente, os parâmetros representam informações opcionais que permitem customização das requisições, habilitando funcionalidades como filtragem ou paginação. Por exemplo, `/usuarios?ativo=true` filtra por usuários ativos, enquanto `/usuarios?ativo=true&pagina=2` acessa a segunda página da lista de usuários ativos.

Códigos de status

A sintaxe dos códigos de status é expressa por números de três dígitos — 200, 404, 500. A semântica desses códigos explica o resultado da operação, como "200 OK" indica que a requisição foi bem-sucedida, "404 *Not Found*" significa que o recurso não foi encontrado e "500 *Internal Server Error*" aponta erros no servidor.

Dados

A sintaxe dos dados abrange a estrutura do corpo da requisição e da resposta, geralmente formatada em JSON ou XML. Os tipos de

dados podem incluir *string*, *number*, *boolean*, *array*, *object*, além de formatos específicos como data, hora, email. A semântica exige correspondência entre o dado e seu valor. Por exemplo, é esperado que o campo `nome` contenha um nome válido e `preço`, um valor numérico positivo. A estrutura dos dados deve estar alinhada com o modelo de domínio, como o objeto `usuario` possui propriedades relacionados a ele, `nome` e `email`.

Esses princípios têm o intuito de tornar APIs REST padronizadas, fáceis de usar, reduzir a ambiguidade e assegurar consistência. A aplicação apropriada da sintaxe e semântica nos componentes moldam a usabilidade, no sentido de como os devs interagem e percebem a API.

18.2 Heurísticas de usabilidade no design de APIs

A usabilidade de API tem o mesmo sentido que a usabilidade de produtos *customer-facing*: a facilidade com que os usuários realizam tarefas. Só tem um probleminha. Devido à falta de conhecimento técnico, não temos condições de opinar sobre uma abordagem adotada e perceber se é fácil ou difícil para um dev usar. Apesar disso, se você souber o que observar, conseguirá avaliar minimamente, quase como acontece com UI.

Sabendo que API nada mais é do que *interface*, podemos nos apoiar nas *10 Heurísticas de Usabilidade para Design de Interface de Usuário* de Jakob Nielsen (2020). Olhe só como as heurísticas consagradas na criação de interfaces de usuário (UI) se correlacionam com as convenções de sintaxe-semântica e como isso se traduz em APIs intuitivas e eficientes:

1. Visibilidade do status do sistema

Mantenha o dev informado com *feedback* rápido e fácil de entender sobre o que está acontecendo. Uma API deve sempre indicar o status das operações requisitadas. Você consegue isso utilizando códigos de status HTTP correspondentes ao tipo de problema. Além disso, é possível incluir mensagens explicativas para o dev entender o que ocorreu e como corrigir. Isso também ajuda as pessoas a aprenderem conforme interagem, o que torna o processo previsível conforme elas avançam.

2. Correspondência entre o sistema e o mundo real

Se você quer prover uma experiência intuitiva, fácil de aprender e de lembrar, use palavras, expressões, conceitos e siga padrões e convenções que os devs já conhecem. Devs familiarizados com REST API esperam que o método `GET` retorne dados, não que os salve. É esperado que um *endpoint* chamado `/customers` tenha informação de clientes e um *endpoint* `/products`, de produtos. Se o usuário não tem ideia do que será retornado com base no nome do *endpoint*, significa que a interface não está intuitiva.

Organize a estrutura de forma lógica, tornando a interação natural e intuitiva. Adicione exemplos e casos de uso reais na documentação para ilustrar o funcionamento da API e facilitar o entendimento.

3. Controle e liberdade para o usuário

Fazer coisas por engano é normal (quem nunca?). Ofereça aos usuários uma maneira fácil de desfazer ações acidentais, sem complicações ou processos longos. Os usuários da API devem ser capazes de cancelar requisições em andamento ou reverter operações. Um exemplo prático seria fornecer um endpoint adicional, como em `/clientes/{id}/restaurar`, que permitiria recuperar o cliente excluído. Essa funcionalidade garante aos devs liberdade de experimentar à medida que aprendem, sem medo de cometer erros irreversíveis.

4. Consistência e padrões

Design inconsistente dificulta a compreensão e o uso da API. A cada variação que aparece, os devs são forçados a aprender algo novo, tornando a adoção mais lenta. O design consistente, por outro lado, torna as coisas previsíveis. Quando o dev aprende algo, o restante da API pode ser inferido e, assim, ele não precisa consultar a documentação constantemente.

Para isso, respeite as convenções de sintaxe e semântica e mantenha a consistência na estruturação e nomenclatura de todos os componentes, tanto internamente quanto em relação a outras interfaces. Lembre-se de que a documentação também precisa ser consistente com o comportamento da API.

Devs que conhecem o funcionamento do protocolo HTTP já sabem o comportamento dos métodos. Eles terão que conhecer os recursos da sua API, mas a mecânica das operações já é conhecida por eles. Então, se a sua API seguir as mesmas convenções do estilo da API, outras de mercado ou internas da empresa, os devs saberão como usá-la, minimizando o que eles têm que aprender especificamente para sua API.

"*Uma API deve ser um pouco como um carro. Todo carro tem volante, pedais de freio e acelerador. Você pode achar que as luzes de perigo, a liberação do porta-malas ou o rádio são ligeiramente diferentes, mas é raro que um motorista experiente não consiga descobrir como dirigir um carro alugado*" (JACOBSON; BRAIL; WOODS, 2011; tradução livre).

5. Prevenção de erros

Diferente de uma interface visual, é improvável que os devs consigam usar uma API sem, pelo menos, uma breve visita à documentação. Mas, o que os devs querem mesmo é usar a API o mais rápido possível, mesmo que na base da tentativa e erro. Por isso, o tratamento de erros é uma parte crítica de DX, pois "*os erros*

se tornam uma ferramenta fundamental que fornece contexto e visibilidade sobre como usar uma API" (GOOGLE CLOUD, 2018).

Melhor do que boas mensagens de erro é criar um ambiente onde é fácil fazer as coisas certas, e difícil cometer erros. Isso pode ser alcançado fornecendo aos usuários da API a chance de confirmarem ações antes de finalizá-las.

Por exemplo, para evitar a exclusão acidental de todos os dados de um banco, poderia ser exigida uma confirmação explícita antes de executar a operação. Imagine uma requisição `DELETE` para o *endpoint* `/clientes`. Se executada, deletaria toda a base de clientes. Para evitar uma solicitação acidental, a API não executaria a ação imediatamente. Uma segunda requisição para `/clientes/exclusao-confirmada` seria necessária. Apenas após essa confirmação, a API executaria a exclusão de todos os clientes. Esse procedimento de duas etapas assegura que exclusões em massa sejam decisões deliberadas e conscientes, reduzindo o risco de incidentes que poderiam resultar em perdas graves.

6. Reconhecimento em vez de memorização

Reconhecimento consiste na nossa capacidade de identificar um evento ou informação por ser familiar; *memorização* refere-se a resgatar da memória. Para que APIs sejam facilmente reconhecidas e reduzam a carga cognitiva, basta se ater às convenções de sintaxe e semântica, mas também forneça documentação completa e atualizada, com exemplos práticos de requisições e respostas.

Utilize nomes descritivos, que representam o significado, para os componentes e mantenha informações importantes visíveis e fáceis de encontrar, para que os devs não precisem memorizá-las de uma parte para outra da jornada.

7. Flexibilidade e eficiência de uso

Crie atalhos para usuários experientes, mas mantenha a interface simples e intuitiva para iniciantes; permita a personalização de ações

frequentes. Você provê mais flexibilidade e eficiência quando suporta diferentes formatos de dados (por exemplo, JSON e XML) e permite customização das requisições com parâmetros opcionais, como filtrar e paginar resultados para lidar com grandes volumes de dados.

Diferentes maneiras de cumprir uma tarefa proporciona flexibilidade, mas também complexidade. O estudo *An Empirical Study of API Usability* (PICCIONI; FURIA; MEYER, c2023) mostrou que devs mais experientes apreciam a possibilidade de escolhas, enquanto os menos experientes muitas vezes podem se sentir confusos. "*Fica claro, porém, que, mesmo para desenvolvedores experientes, a escolha só é positiva quando as opções são realmente complementares, em vez de serem apenas complicações desnecessárias do design*" (PICCIONI; FURIA; MEYER, c2023; tradução livre).

8. Design estético e minimalista

Um design estético e minimalista concentra-se no essencial, evitando distrações com elementos irrelevantes e desnecessários. Mantenha a API enxuta, oferecendo apenas *features* necessárias. Evite *endpoints* redundantes ou pouco usados e parâmetros desnecessários que acabam confundindo os devs. Documente apenas informações relevantes e com exemplos práticos.

Parece óbvio que interfaces menos complexas são mais fáceis de utilizar. No entanto, o artigo *Improving API Usability* afirma que não encontraram "*correlação entre o número de elementos em uma API e sua usabilidade, desde que tivessem nomes apropriados e estivessem bem-organizados*"; e complementa que fornecer "*diferentes conjuntos de parâmetros pode melhorar a usabilidade*" (MYERS; STYLOS, 2015; tradução livre). Mas aqui vale se atentar ao perfil dos devs usuários, já que os menos experientes podem se perder diante de variadas opções.

9. Ajude os usuários a reconhecerem, diagnosticarem e recuperarem-se de erros

Apesar da sintaxe apropriada, mensagens de erro como `400 Bad Request` não são muito úteis, apenas confirmam que o erro está na requisição do cliente, mas não fornecem o que exatamente deu errado. A documentação deve esclarecer as condições que desencadeiam erros e as próprias mensagens de erro podem conter informações úteis.

Você aprimora a semântica por meio de mensagens que explicam o problema e oferece soluções, por exemplo, "Erro 400: O campo email é obrigatório. Inclua um email válido e tente novamente.". Sem isso, devs ficam sem meios de recuperação, e você desperdiça a oportunidade de guiar a pessoa exatamente no momento em que ela está mais propensa a ler e seguir as instruções.

Mensagens de erro concisas, descritivas e em formato legível por pessoas (não máquinas) podem ser usadas como instrumentos de aprendizado sobre o uso da API. Quando as orientações não podem ser incluídas de forma sucinta, você pode adicionar o link para a documentação na mensagem, assim os devs resolvem o problema por conta própria.

10. Ajuda e documentação

Documentação insatisfatória — imprecisa ou incompleta — pode levar a falhas de usabilidade. É o que revela o estudo *An Empirical Study of API Usability* (PICCIONI; FURIA; MEYER, c2023). A pesquisa *2022 State of the API Report* (POSTMAN, 2022) revela três pontos relevantes sobre esse tema:

1. 64% dos respondentes levam em consideração a documentação antes de integrar a uma API, perdendo somente para performance (72%), segurança (71%) e confiabilidade (68%). A usabilidade da API ficou em sexto lugar (51%);
2. 55% das pessoas entrevistadas afirmam que a falta de documentação é o principal obstáculo no consumo de APIs.

Apenas 3% delas afirmam que as APIs com as quais trabalham são "muito bem documentadas";
3. Sobre o que pode ser melhorado nas documentações, os top 5 itens foram: documentação atualizada (57%), exemplos de código (55%), exemplos melhores (53%), documentação padronizada (51%) e casos de uso do mundo real (45%).

Mesmo em uma interface intuitiva que minimize a necessidade de explicações, sem documentação, é improvável que devs se saiam bem no uso, pois não saberão como acessá-la, os nomes dos recursos e a estrutura das requisições. Então, como a API pode ser bem-sucedida como produto se os devs não sabem como começar a usá-la?

No meu entendimento, documentação é parte do pacote que torna a plataforma um produto. Produto mal resolvido gera suporte. Os devs usuários não conseguirão implementar as soluções de modo *self-service*, com autonomia, sem documentações eficazes. Por consequência, o time da plataforma dedicará tempo demais em suporte, o que o impedirá de trabalhar na própria plataforma — entrave que documentações podem eliminar.

Por isso, tão importante quanto uma boa interface é uma boa documentação. Ambas moldam a experiência do dev e isso se traduz em valor estratégico. Sem uma documentação suficientemente boa, a API será mais difícil de aprender e utilizar, e se o uso é imposto aos times — comum quando a plataforma é interna —, levará mais tempo de desenvolvimento e para lançamento, além de maior probabilidade de bugs, portanto, terá necessidade de suporte do time da plataforma quase como quando não há documentação.

A verdade é que criar e manter documentações pode ser demorado e entediante, principalmente se forem estáticas, em que a cada atualização da API a documentação também precisa ser atualizada manualmente, o que costuma ser esquecido ou deixado de lado.

Para tornar o processo mais produtivo e conveniente, existem ferramentas no mercado que geram a documentação automaticamente, com interface visual interativa, e a mantêm atualizada — será apresentado em detalhes na *Prototipagem*. Por ora, vamos explorar alguns componentes na *Ideação* partindo dos princípios que vimos até aqui.

18.3 Ideação

Vamos imaginar o seguinte contexto fictício: uma grande ONG que resgata cães e gatos possui um site chamado *Petflix*. Depois de prover os devidos cuidados, os pets são disponibilizados no site para adoção. Potenciais adotantes podem navegar e acessar o perfil de todos os pets sem necessidade de login. Para adotar, é necessário preencher um formulário que é avaliado pela equipe da ONG, que aprova ou reprova a adoção.

A ONG pretende plataformizar alguns recursos internos e nas conversas com *stakeholders* internos — na fase de *Empatia* — chegou-se aos seguintes **casos de uso** para atender a dois perfis de usuários finais:

1. **Colaboradores**: pessoas que trabalham na ONG
    - Cadastram pets, e não é preciso login para isso — apenas manter a didática simples; no mundo real, o ideal é que mesmo os usuários internos façam login por questões de segurança;
    - Acessam a lista com todos os pets cadastrados.
2. **Adotantes**: pessoas interessadas em adotar pets
    - Acessam a lista com pets disponíveis para adoção;
    - Preenchem o formulário de interesse em adotar determinado pet.

A partir disso, é possível pensar em como será a modelagem da API, o que pode ser feito pela pessoa responsável pelo design da API ou pode ser um trabalho de cocriação, com pessoas do próprio time da plataforma e times clientes (por que não?). A abordagem aqui é sob a perspectiva de cocriação.

Wireframe de baixa fidelidade

Algumas pessoas são convidadas para a sessão de criação do *wireframe*. Relembrando, é o esboço das telas do produto ou app que demonstra como os elementos se relacionam e como são estruturados. Isso traz visibilidade da hierarquia das informações, funcionalidades, possíveis ações e comportamentos dos usuários. O que você verá agora tem o mesmo objetivo, por isso chamo de *wireframe*.

A sessão de criação do *wireframe* começa com o compartilhamento do que foi descoberto até então e as hipóteses de casos de uso em forma de mapa mental, como o da figura a seguir:

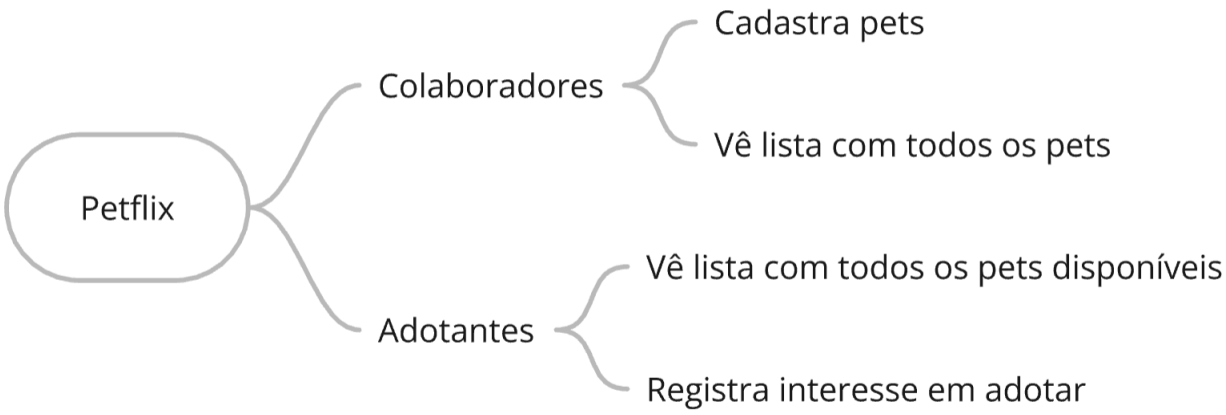


Figura 18.1: Casos de Uso

Os participantes percebem então mais casos de uso e, organicamente, se torna um *brainstorm*. O mapa mental é incrementado ao longo da dinâmica e toma a forma da figura seguinte:

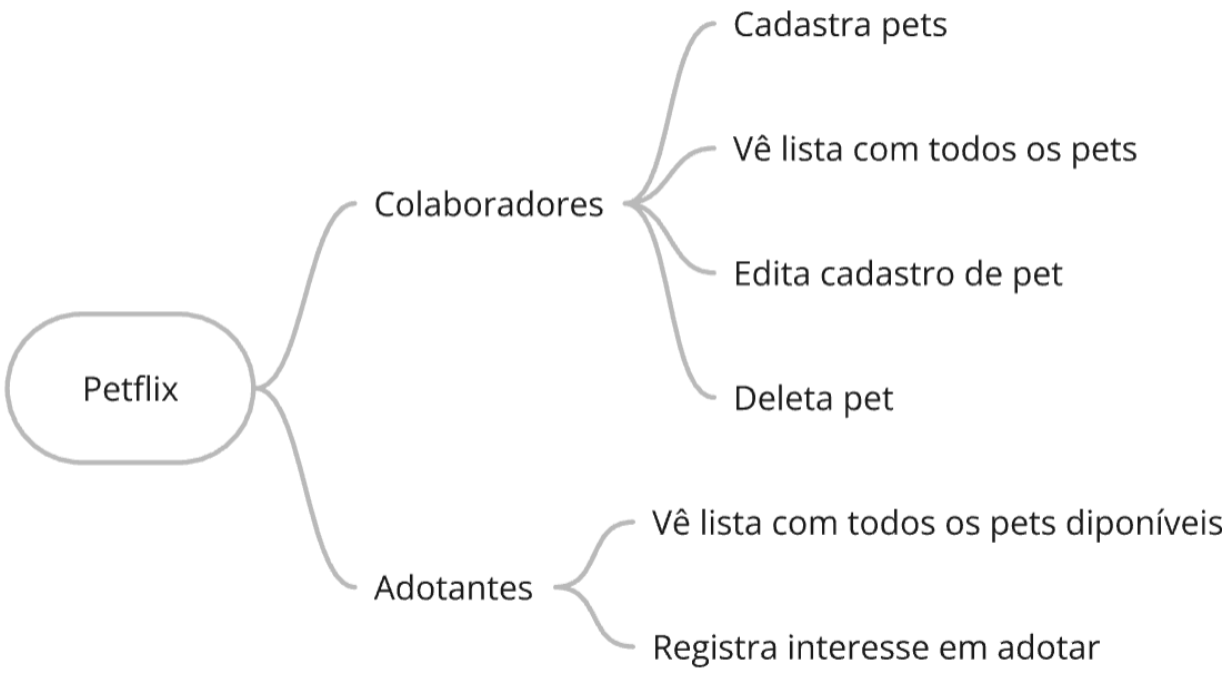


Figura 18.2: Casos de Uso

Em resumo, sempre há a possibilidade de inserir algum dado errado ou de atualizar alguma informação do pet, por isso deve ser permitido alterar as informações. Além disso, talvez queiram deletar o cadastro de determinado pet.

Veja que os verbos "cadastra", "vê" e "edita" são **ações** sobre certa coisa. Essa *coisa* é tecnicamente chamada de **objeto** ou **entidade**. Cada objeto é composto por **atributos**. Nesta situação, o que sofre ações é o objeto *pet*. A próxima imagem mostra os atributos citados na dinâmica (fictícia), considerando o contexto da ONG.

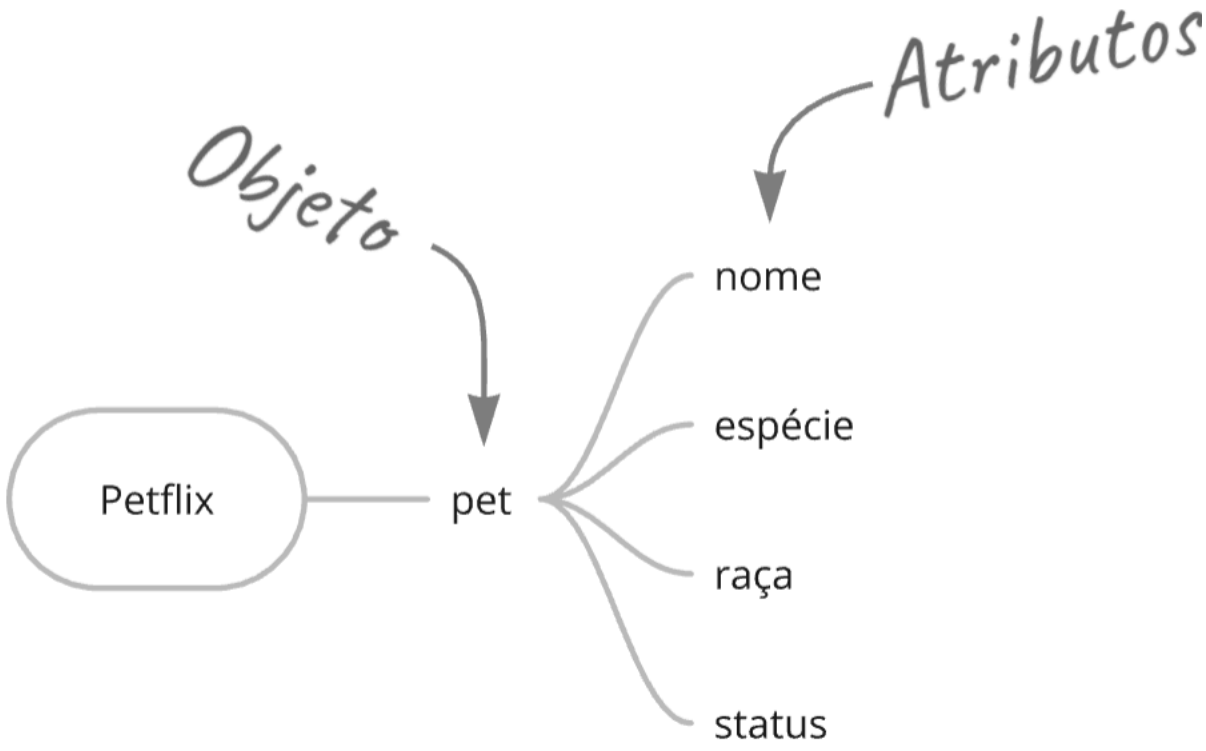


Figura 18.3: Objeto e atributos

Pet é um objeto, e tudo sobre pets é um **recurso**. O recurso é exposto por meio de um ***endpoint*** e, seguindo as convenções de API REST, *endpoints* devem ser **substantivo no plural** — `/pets` — como vimos na heurística consistência e padrões. Além disso, a nomenclatura de cada elemento procura obedecer à correspondência entre o sistema e o mundo real, que serão validados nos testes com os devs.

Cada pet é único; mas, e se tiverem o mesmo nome, como diferenciá-los? É necessário um identificador exclusivo para cada um, o `id`, bem como é preciso definir o tipo de dado para cada atributo. Ter um exemplo real da **formatação dos dados** ajuda a tangibilizar a hierarquia (quando há) e o que é esperado como valor de cada dado, como exibe o mapa mental adiante.

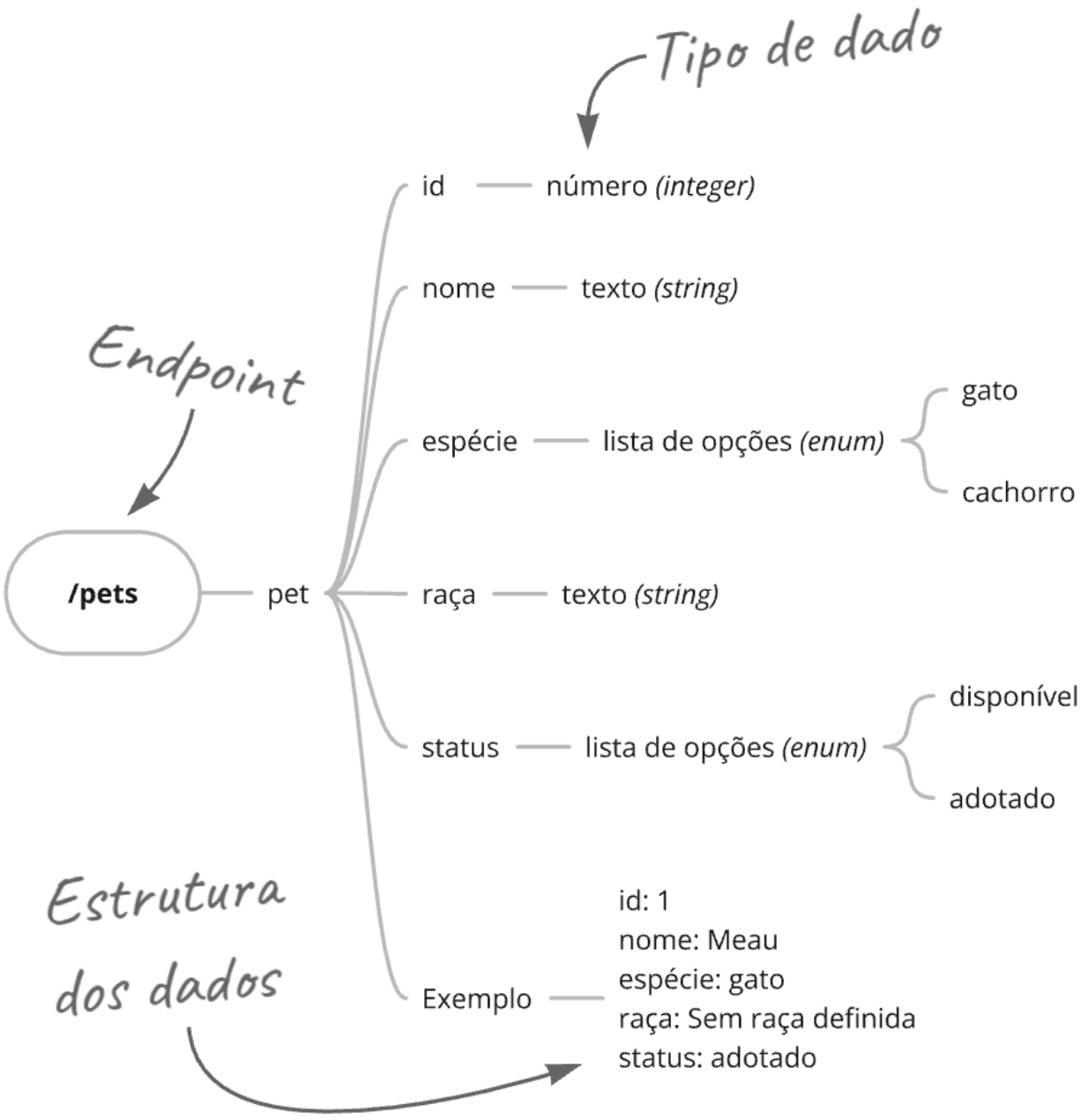


Figura 18.4: Objeto e atributos

Depois de definir os atributos, os participantes da sessão chegaram a algo como a próxima imagem ao final das definições sobre as ações CRUD para, no momento oportuno, associar aos respectivos **métodos HTTP** para manipular o objeto.

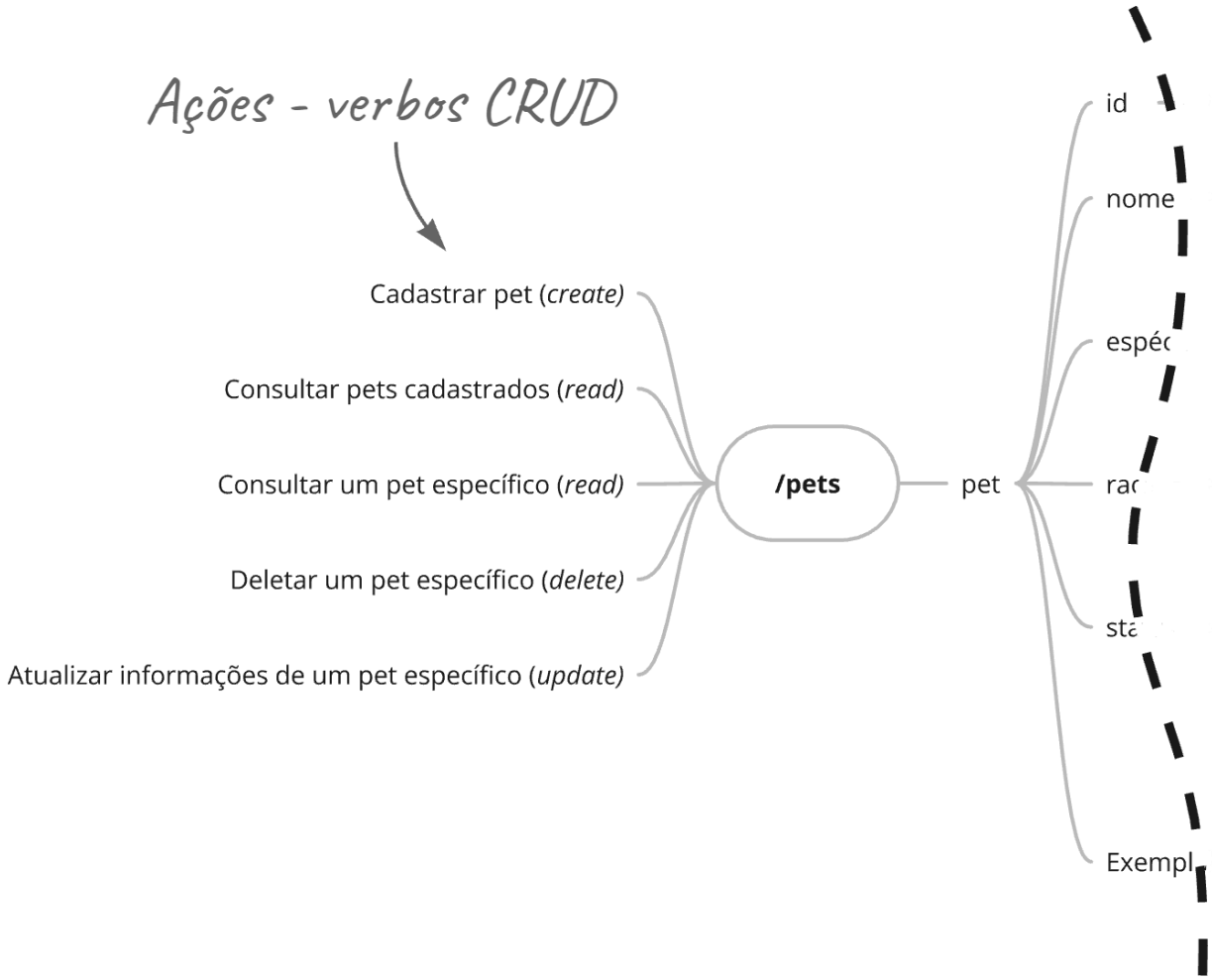


Figura 18.5: Ações — CRUD

Depois da ação, o dev usuário precisa saber se a requisição foi bem ou malsucedida, de acordo com a heurística de visibilidade do status do sistema, certo? A próxima figura ilustra o **código de status** e a **resposta** a ser retornada para cada situação, estabelecido ainda na sessão de criação, em consonância com algumas heurísticas de usabilidade.

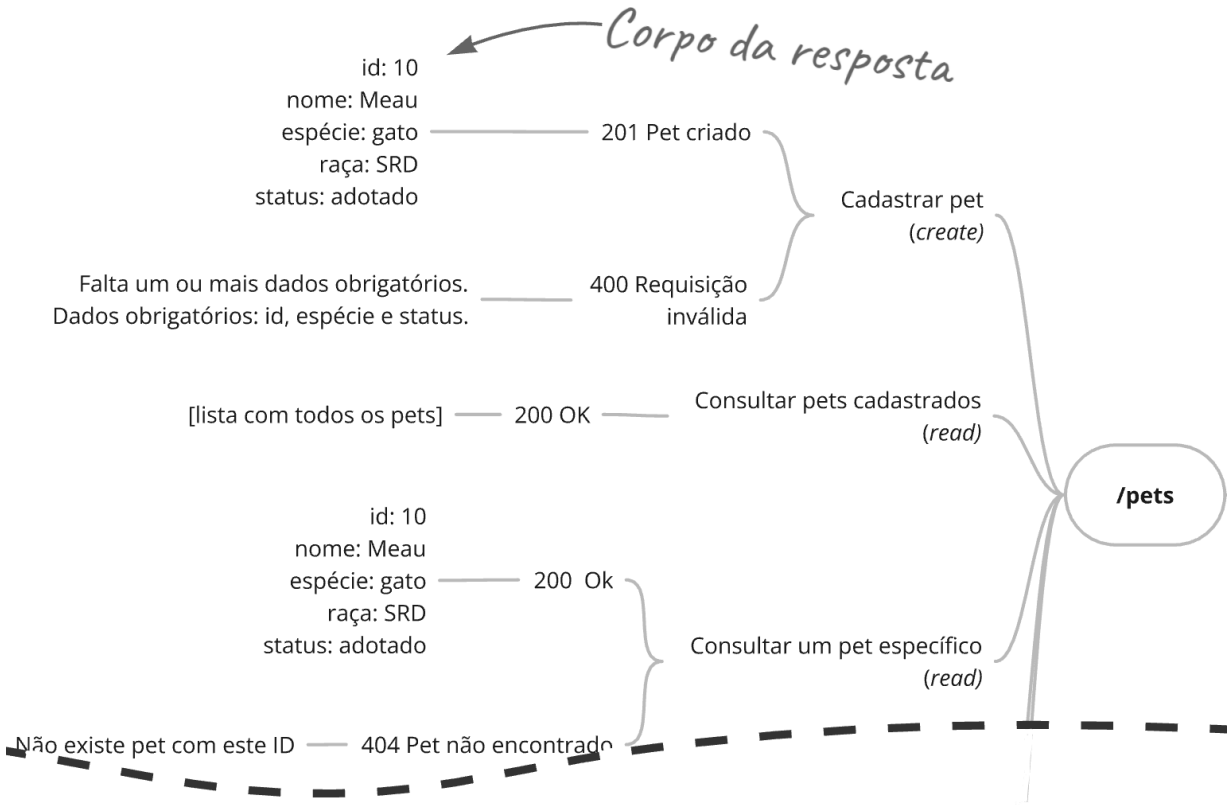


Figura 18.6: Código de retorno e mensagem de resposta

Apesar de a ideação ter começado com conceitos HTTP em mente, note que está em uma linguagem genérica, que descreve os elementos sem *techniquês*. Você, como PM, consegue contribuir até aqui com tranquilidade, não? Nem é preciso conhecer os códigos de status — foi demonstrado assim apenas para você ir se familiarizando.

Elaborar o design da API de modo genérico permite que a ideia possa ser implementada em qualquer estilo de arquitetura, que pode ser escolhida depois do wireframe. Outro ponto a se observar é que a prática foi apresentada em forma de mapa mental, mas poderia ser em tabela ou texto. Em formato texto, poderia ser algo assim:

Atributos

- pet:
  - id: numérico;
  - nome: texto;
  - espécie: lista com as opções gato e cachorro;
  - raça: texto;
  - status: lista com as opções disponível e adotado.

Ações:

- /pets
  - Criar pet (*create*):
    - 201 Pet criado: retorna todos os dados do pet no corpo da resposta;
    - 400 Requisição inválida: retorna mensagem "*Falta um ou mais dados obrigatórios. Dados obrigatórios: id, espécie e status.*".
  - Consultar pets cadastrados (*read*);
  - ...

Este exemplo, apesar de simples e de parecer que não haverá interpretações ambíguas, não deixa de conter hipóteses sobre o que se acha que funcionará. No mundo real, as definições são mais difíceis, mas é preciso começar de algum lugar. Boa parte do trabalho de ideação é simplificado quando a empresa segue um *styleguide*. Ainda assim, é provável que haja elementos que requerem sintaxe e semântica específicas do domínio. Nesse cenário, vale a pena usar APIs públicas que tratam do mesmo tema como referência. Por exemplo, para uma plataforma que envolve pagamentos você pode usar a Stripe como inspiração, especialmente para nomenclaturas. Apesar de ser uma API consagrada no mercado, não é garantido que os elementos se enquadram no contexto da sua empresa, por isso você precisará testar as hipóteses

do design. Um *wireframe* bem elaborado serve para facilitar a criação do protótipo.

18.4 Prototipagem

Ao final do processo de design de UI é comum que seja gerada uma documentação com os componentes, as interações, a representação dos estados (vazios, preenchidos, sucesso, erro etc.) e o protótipo que simula as interações, o fluxo e o resultado. Podemos fazer o mesmo para APIs. Na etapa de prototipagem, o que foi definido no *wireframe* receberá uma aparência mais próxima da real sobre: *endpoints*, métodos, formatação dos dados das requisições e das respostas.

Há duas maneiras de se chegar a este tipo de protótipo:

1. Definir o contrato por meio de uma linguagem que expressa a especificação da API;
2. Sem recorrer a linguagem alguma para definição do contrato ou escrever uma linha de código sequer.

Ambas as possibilidades geram uma documentação visual e interativa para ser compartilhada com os usuários da API e pode ser incrementada com exemplos, descrições, mensagens de erro, entre outras coisas.

Pessoas mais sêniores de engenharia dizem que é simples e rápido "codar" o contrato — especialmente se comparado ao desenvolvimento do back-end. No entanto, isso pode tomar tempo quando exige inúmeras mudanças. Por isso, a segunda alternativa pode ser interessante, porque é possível iniciar discussões com os times sobre nomenclaturas e estruturas, por exemplo, antes de "codar" o contrato da API.

A opção de testar algumas ideias sem a necessidade de codar, além de agilidade na validação das hipóteses, é um processo que nós mesmos, PMs sem *background* técnico, conseguimos executar. Não acredita? Então, confira como você pode fazer isso no Postman.

Criação do mock server no Postman

Você lembra do Postman do capítulo anterior, certo? Lá, a API foi testada acessando *endpoints* de um servidor real da Swagger Petstore, com dados reais. Aqui, será acessado um *mock server* (servidor simulado), uma ferramenta que simula o comportamento de um servidor. Para criar a simulação, é necessário configurar os *endpoints* que o *mock server* deve simular, junto das respostas correspondentes para cada tipo de requisição (GET, POST, PUT e DELETE). Então, quando alguém envia requisições, o *mock server* responde como um servidor real, mas com dados pré-configurados.

Para testemunhar esse acontecimento na prática, entre na sua conta e clique em "*Mock Servers*" no menu à esquerda e, em seguida, no link "*Create Mock Server*", onde mostra a próxima figura.

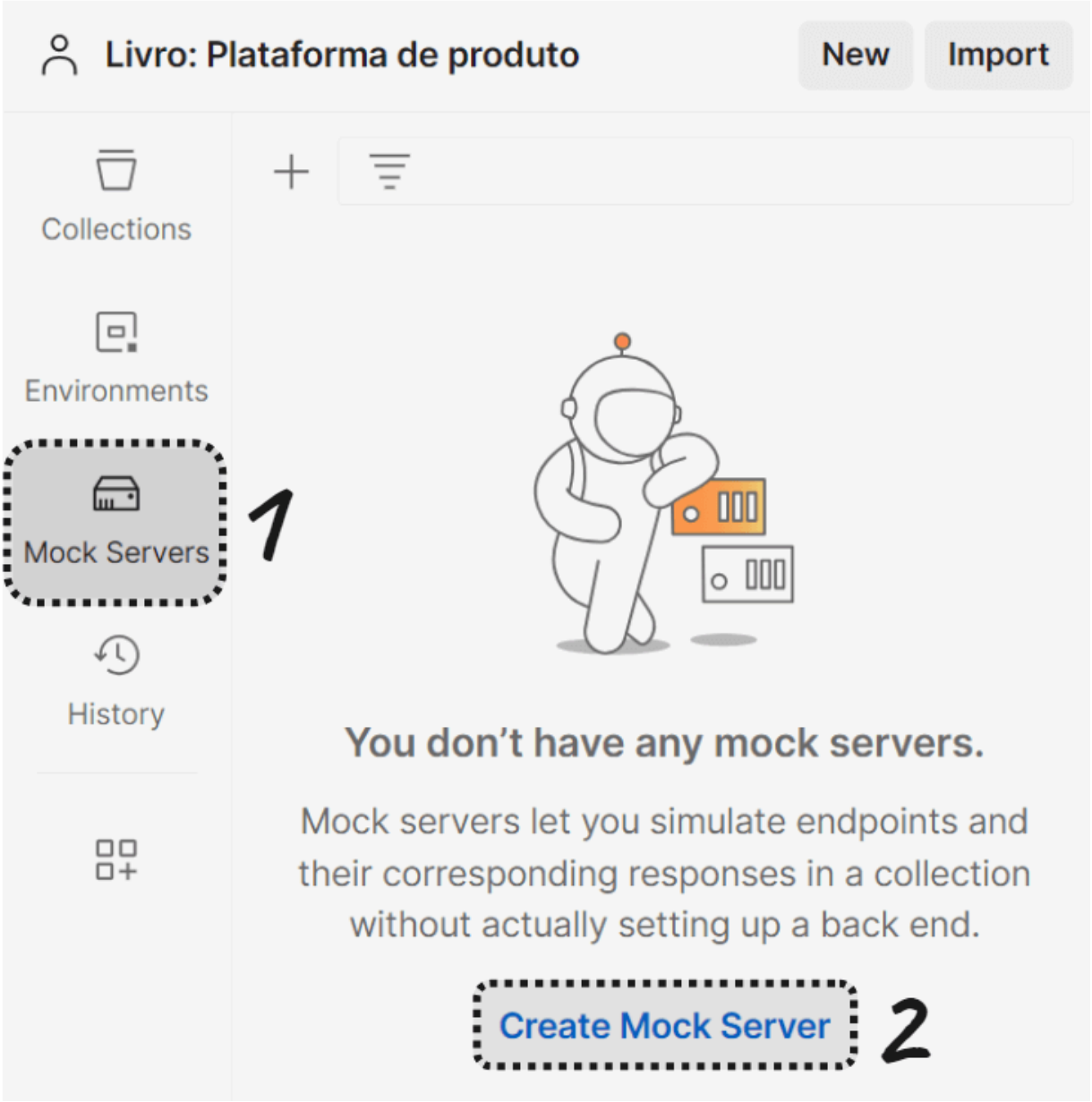


Figura 18.7: Criando o mock server.

Abrirá uma nova aba em que há duas coisas que precisamos configurar:

1. ***Path***: o caminho de requisição. Ele será parte da URL do *endpoint* e, de acordo com nosso *wireframe*, seria `/pets` ;
2. ***Response Body***: o corpo da resposta, o conteúdo do que será retornado quando feita a requisição. Vamos apenas colocar um

texto simples em formato "chave: valor" — `nome: Meau`, o exemplo incluído no *wireframe*.

Isso feito, basta clicar no botão "*Next*".

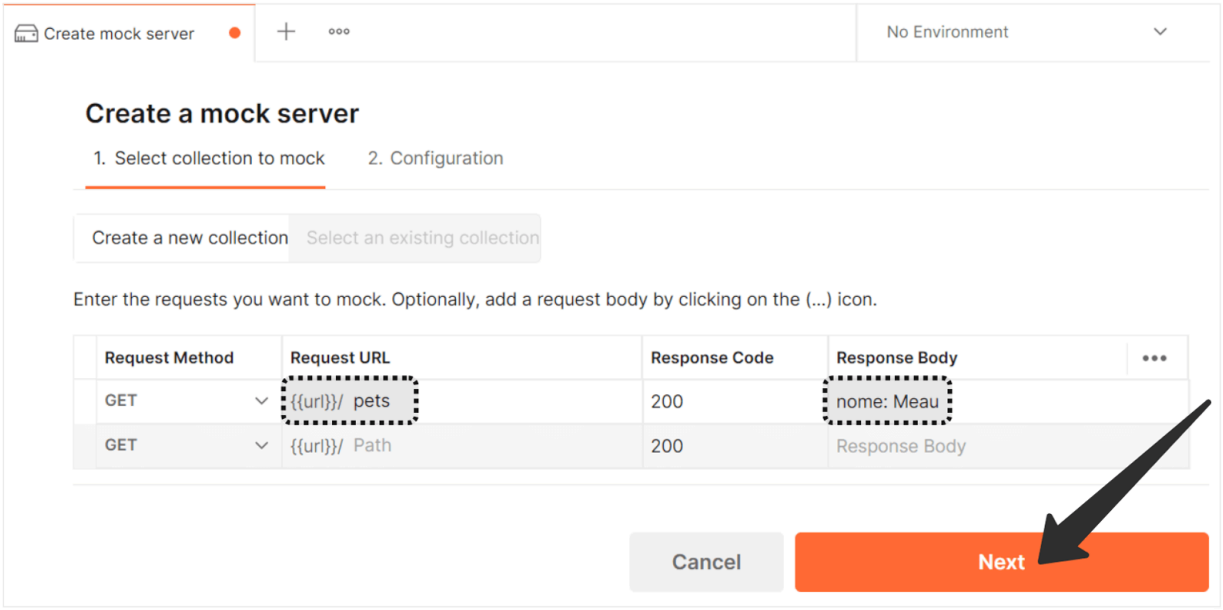


Figura 18.8: Criando o mock server.

Depois disso, abrirá uma tela para mais configurações. Precisamos apenas nomear o *mock server* — "Protótipo Petflix" — e selecionar "*Save the mock server URL as a new environment variable*" (salvar a URL do *mock server* como uma variável de ambiente) — as demais opções de configuração podem ficar como estão. Clique em "*Create Mock Server*".

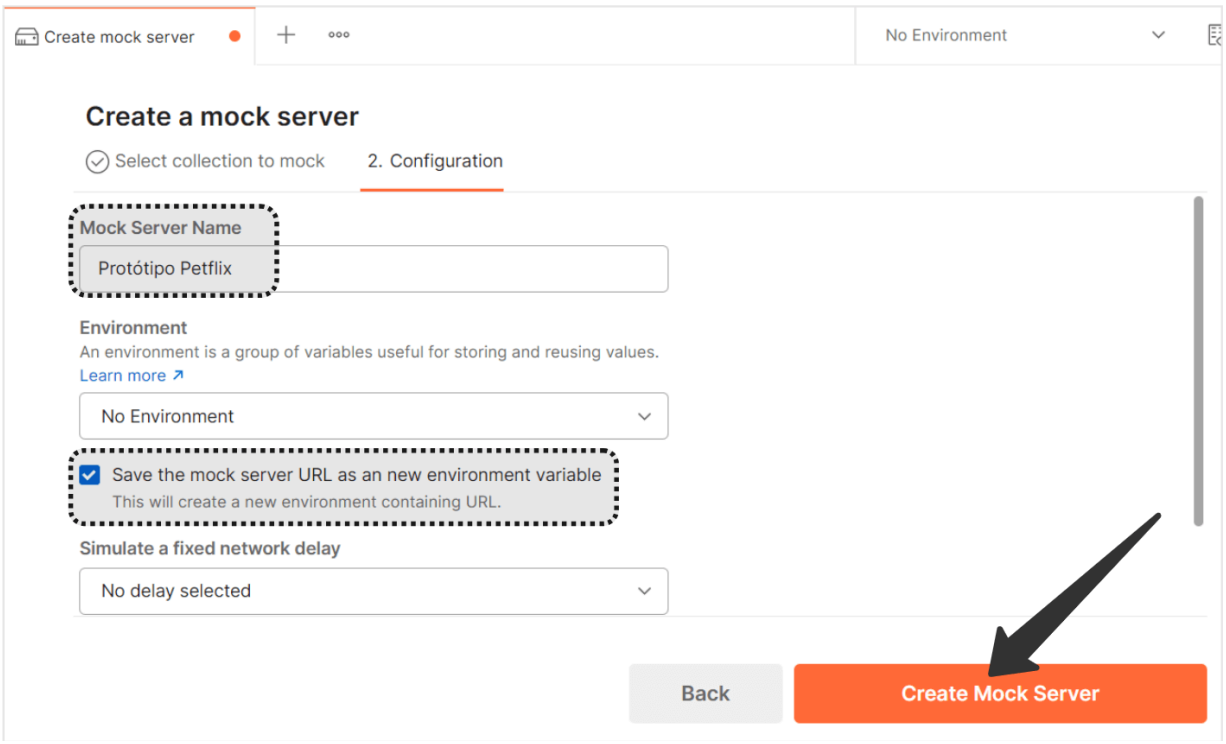


Figura 18.9: Criando o mock server.

A próxima imagem exibe que o Postman criou o *mock* e a respectiva URL à qual serão feitas as requisições e respostas — veremos em funcionamento mais à frente. Vamos alterar uma configuração do *mock server*. Para isso, no topo, clique em "*Edit Configuration*", como indicado pela seta na imagem adiante.

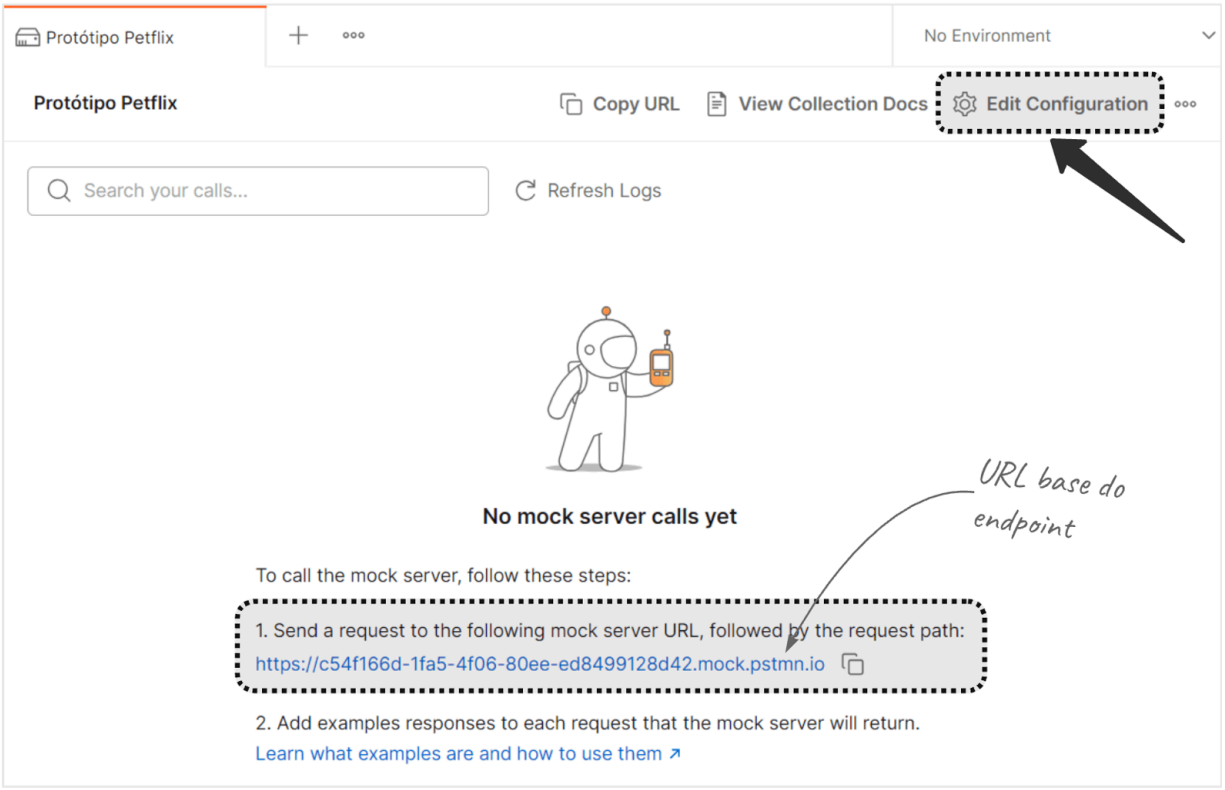


Figura 18.10: Criando o mock server.

No final da tela que se abrirá, vamos selecionar *Request body*. Assim, quando alguém enviar uma requisição com determinado conteúdo ao *mock*, o servidor devolverá a resposta correspondente.

Edit configuration

Mock Server Name

Protótipo Petflix

Collection

Protótipo Petflix

Environment

An environment is a group of variables useful for storing and reusing values.
Learn more ↗

No Environment

Simulate a fixed network delay

No delay selected

☐ Make mock server private

To call a private mock server, you'll need to add an `x-api-key` header to your requests. See how to generate a Postman API key ↗

Response Matching

Choose what response is served by your mock server. Learn more ↗

☑ Request body ⓘ

☐ Headers ⓘ

Update Mock Server Cancel

Figura 18.11: Criando o mock server.

Observe no próximo *print*. Além de criar o *mock*, o Postman criou uma *collection* com o mesmo nome do *mock server*, "Protótipo Petflix", e contém a requisição `GET`. No campo da requisição, `{{url}}` representa a variável que é substituída automaticamente pela URL do *mock* e note que `/pets` já foi adicionado ao final da URL. Se você clicar no botão "*Try*", verá `nome: Meau` no *Body* do campo de resposta e o código de status 200. Esta resposta é aquela que inserimos no começo das configurações.

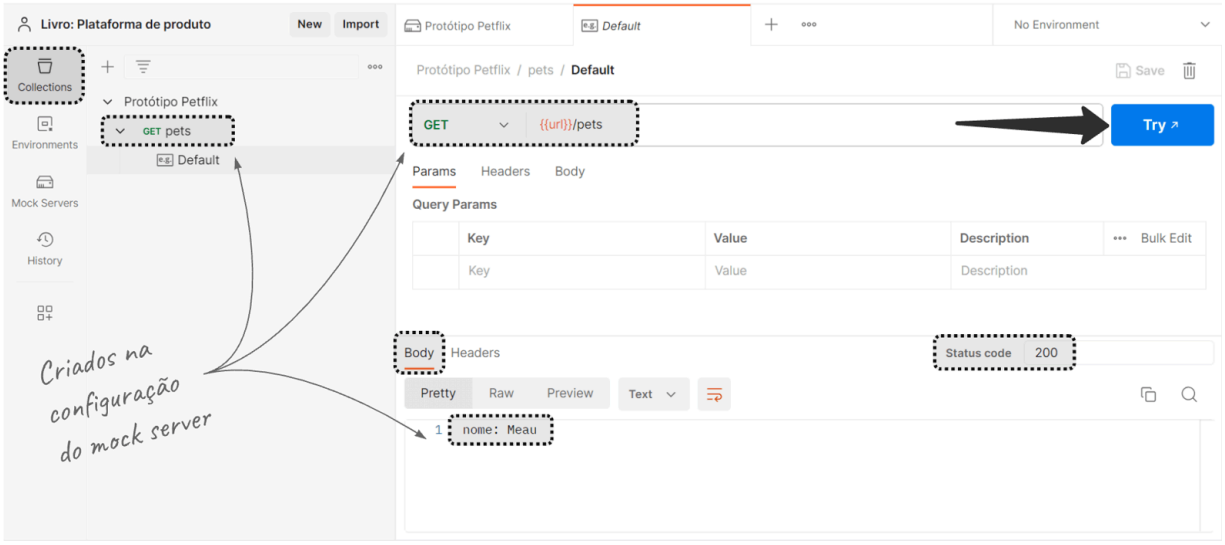


Figura 18.12: Mock server criado

No meu ponto de vista, uma forma de manter mais organizado e evitar que deixemos passar o básico é seguir a sequência CRUD, que em REST seria `POST`, `GET`, `PUT` e `DELETE`. De acordo com nosso *wireframe*, quando a requisição for para criar um registro — `POST` —, o conteúdo da resposta traz todos os dados do pet cadastrado. Então, vamos editar essa requisição-resposta e criar mais algumas.

Requisição e resposta

Uma requisição pode ter mais de um exemplo de resposta, por exemplo, uma para sucesso e outra para erro. Nesse caso, o *mock* compara os exemplos configurados e retorna a resposta que

corresponde à formatação da requisição. Agora, vamos produzir os exemplos de resposta para a requisição de criação de registro de um pet, editando aquele exemplo criado nas configurações do *mock*. Primeiro, mude o método de `GET` para `POST`. Clique no menu *dropdown*, selecione `POST` e depois clique no botão "*Save*", como a imagem seguinte:

Figura 18.13: Alterando o método da requisição.

Na próxima figura é possível visualizar que, no menu, o método mudou para `POST`. Abaixo dele, vemos o exemplo criado no momento da configuração do *mock*, "*Default*". Clique sobre ele.

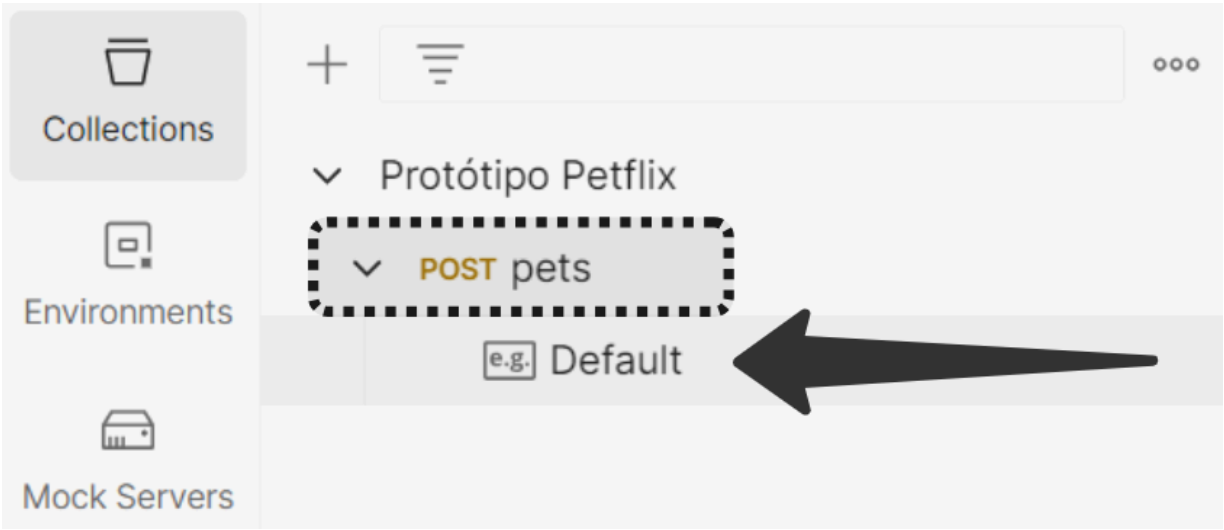


Figura 18.14: Alterando o método da requisição.

Na nova aba que abrirá, mudaremos o nome do exemplo de "*Default*" para "*201 Created*" — poderia ser um nome qualquer, mas o ideal é que seja algo que descreva que o exemplo se refere a sucesso ou erro.

Observe atentamente a figura que segue para configurar a requisição seguindo os passos 2 e 3. Selecione "*Body*", em seguida, "*raw*" e mude para "JSON".

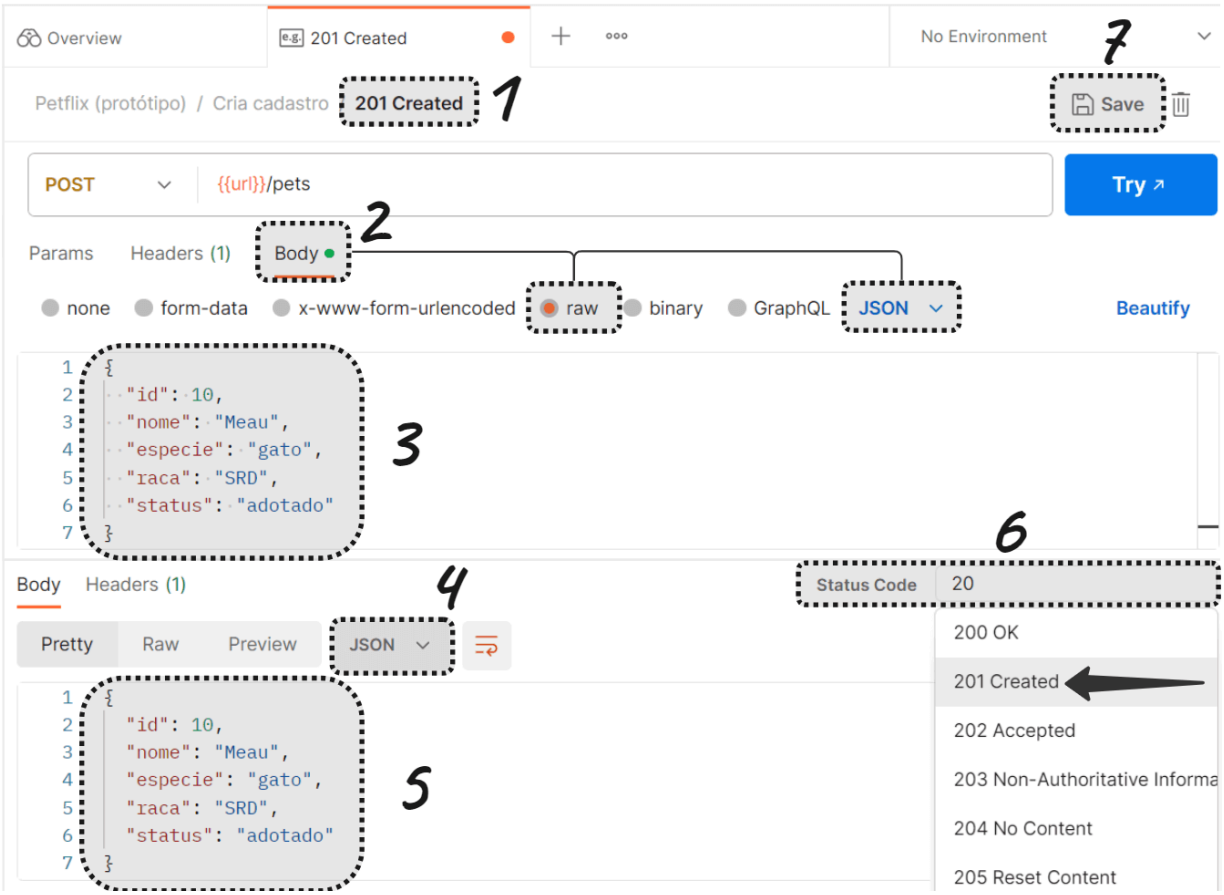


Figura 18.15: Criando exemplo requisição-resposta para sucesso.

Para criar um cadastro de pet, é necessário enviar os dados na requisição. Então, no campo de texto, vamos inserir o exemplo elaborado no *wireframe* da mesma forma que você vê no código adiante. Repare nas chaves no começo e no final, nas aspas em cada chave e em cada valor, e que cada linha é separada por vírgula, exceto a última. Essa é a formação de um código em JSON, linguagem que foi selecionada anteriormente.

```
{
  "id": 10,
  "nome": "Meau",
```

```
  "especie": "gato",
  "raca": "SRD",
  "status": "adotado"
}
```

Em seguida, vamos configurar o exemplo de resposta para esta requisição, de acordo com os passos de 4 a 6. O wireframe indica que retorna os mesmos dados da requisição, como uma maneira de deixar claro que o pet foi cadastrado. No campo "*Body*", mude de "*Text*" para "JSON" e insira o mesmo conteúdo do código anterior no campo "*Body*". Em "*Status code*", selecione `201 Created`, o status padrão para quando algo é criado — ao digitar os números, o sistema fornece as opções de código. Por fim, clique em "*Save*" (passo 7).

Vale lembrar que o conteúdo da resposta foi decidido pelo time da plataforma no *workshop* de design do *wireframe*. Nesse caso fictício, foi decidido retornar os dados da requisição, mas poderia ser uma mensagem como "O pet com ID 10 foi criado". O conteúdo vai depender do que os usuários da API têm de expectativa em receber de retorno (que será validado no teste com os devs usuários). Claro que isso inclui os erros, afinal eles vão querer saber como corrigir, então vamos adicionar um exemplo para esse tipo de cenário.

Posicione o mouse sobre o método, e assim será exibido o ícone para mais ações (três pontinhos) — onde mostra o próximo *print*. Ao colocar o mouse sobre o ícone, o menu se abrirá. Clique em "*Add example*".

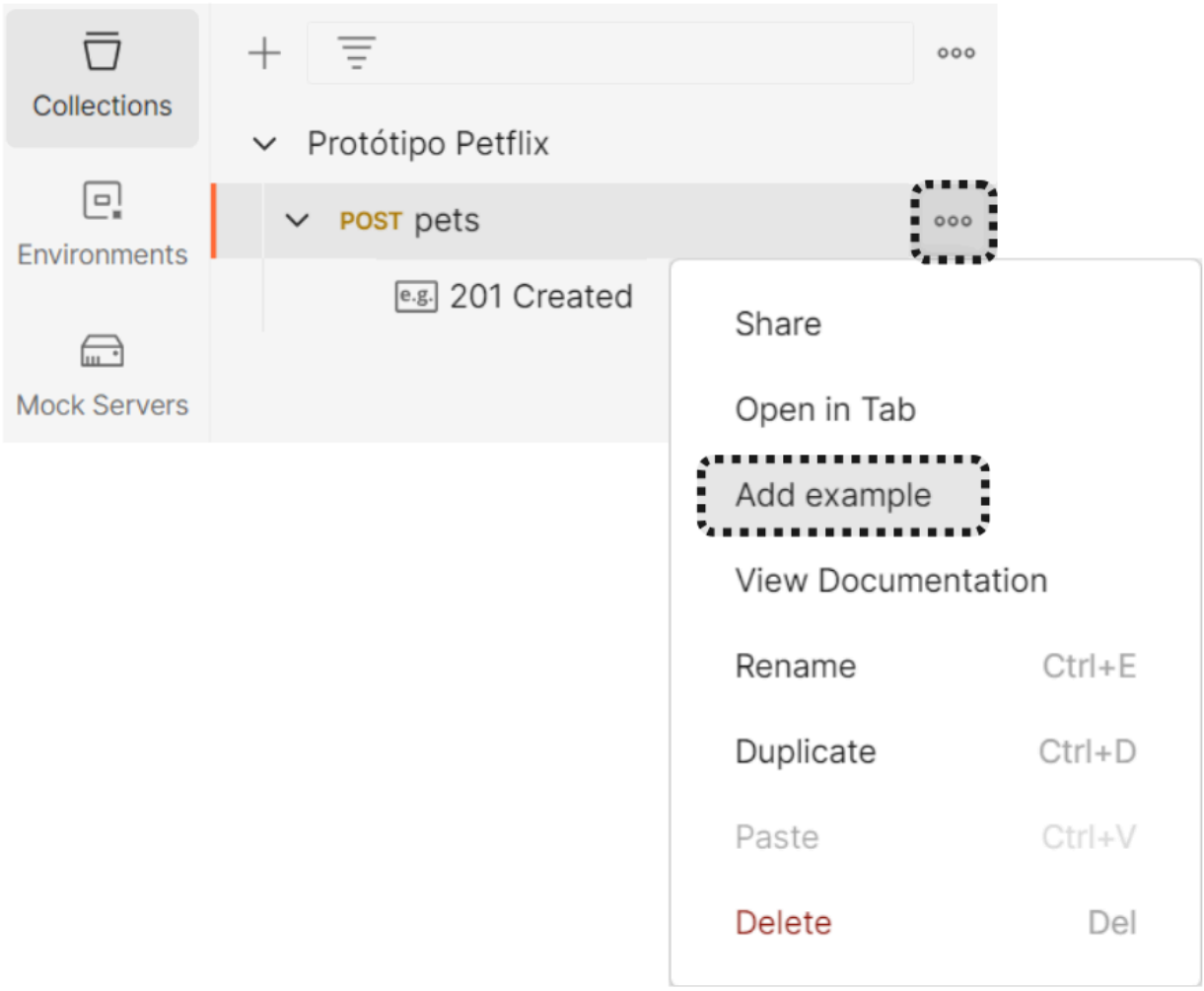


Figura 18.16: Adicionando exemplo.

Para este cenário, vamos supor que foi decidido que o ID, a espécie e o status do pet são atributos obrigatórios para criar um cadastro. Portanto, se faltar um destes dados ou o valor estiver vazio, deve ocorrer um erro.

Siga os passos do *print* que segue. Nomeie como "400 Bad request" e, no "*Body*" da requisição, insira o mesmo conteúdo do exemplo anterior e delete a linha do atributo `id` — certifique-se de que "*raw*" e "JSON" estão selecionados. No "*Body*" da resposta, coloque a mensagem — a mesma definida no *wireframe*. No campo "*Status code*", selecione `400 Bad Request` e lembre-se de salvar o exemplo, clicando em "*Save*".

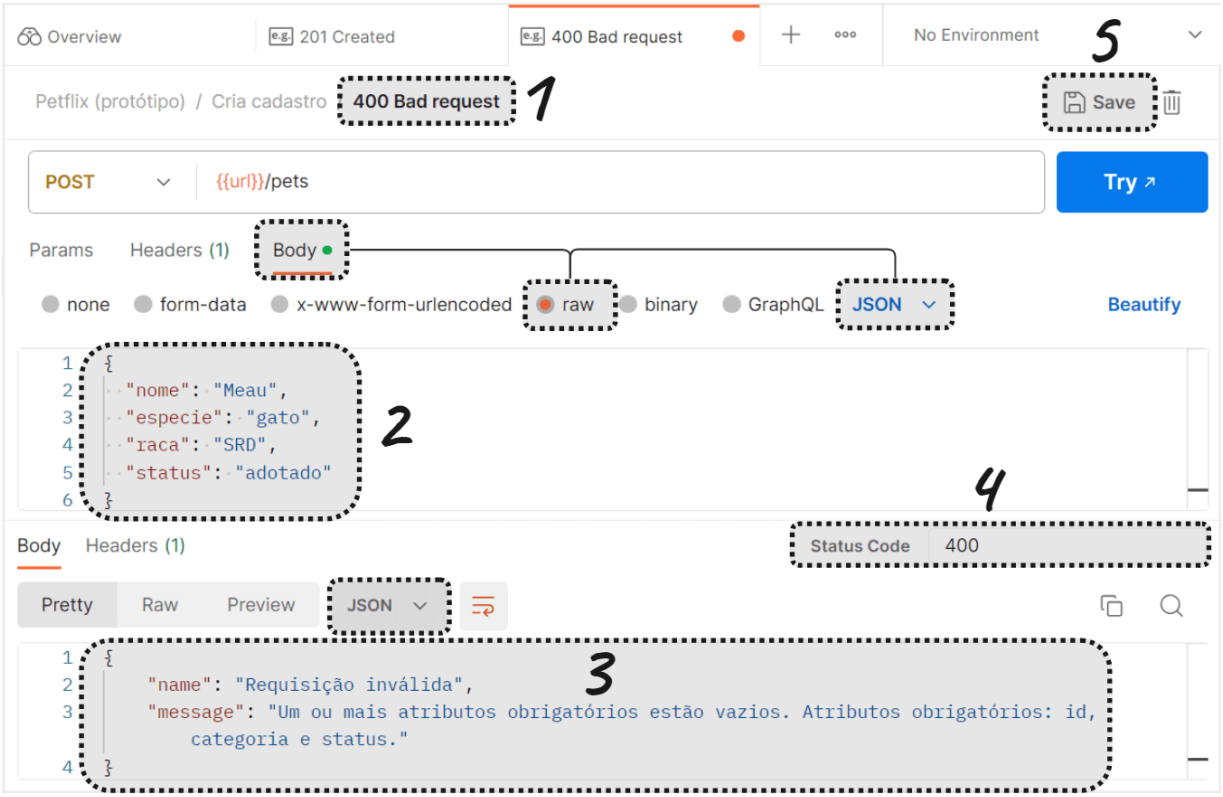


Figura 18.17: Criando exemplo requisição-resposta para erro.

Vamos ver o que acontece quando ocorre uma requisição? No menu à esquerda clique na requisição `POST`. A imagem seguinte mostra a aba que abrirá. Clique em "*Body*", selecione "*raw*" e "JSON". Insira o código exatamente igual ao *body* da requisição que criamos para sucesso (201 *Created*), depois, clique em "*Send*" — é isso o que os devs vão fazer no teste de usabilidade. O Postman retorna a resposta correspondente à operação bem-sucedida: `201 Created` e o conteúdo da resposta no "*Body*". Maravilha! Nosso exemplo está funcionando como esperado.

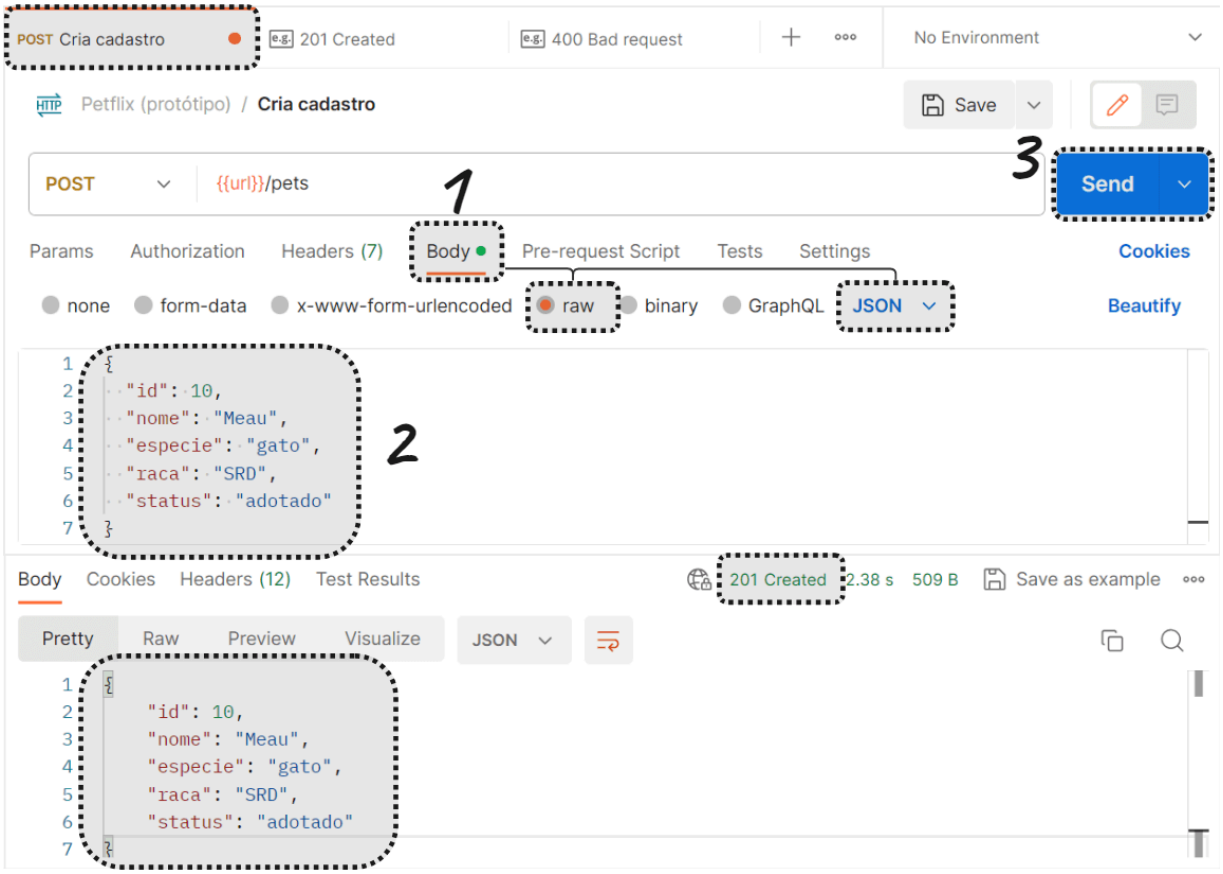


Figura 18.18: Testando requisição de sucesso.

Vejamos como será o teste de usabilidade para o cenário em que não é enviado o ID no *body* da requisição. Para isso, ainda na mesma aba, vamos apenas deletar o conteúdo da linha 2 — `"id": 10,` — e clicar em "*Send*". Veja na imagem a seguir que foram retornados o código de erro `400 Bad Request` e a mensagem que configuramos no *body* da resposta.

Figura 18.19: Testando requisição de erro.

Agora, o que acontecerá se tirarmos outro atributo obrigatório, por exemplo, o `status`? O Postman não sabe que esse é um atributo obrigatório. Nessa situação, ele retorna um erro verdadeiro — que pode ser visto na imagem adiante. Isso acontece porque não criamos o exemplo de resposta sem `status`, portanto o Postman não o encontra — `404 Not Found`. Esse tipo de protótipo é igual àqueles em que o usuário deveria preencher campos de um formulário de cadastro no app, mas os campos vêm preenchidos e o conteúdo é estático. Se você quiser contemplar inúmeras possibilidades, você terá de criá-las. Então, antes dessa fase, planeje bem o que você (e seu time) precisam validar.

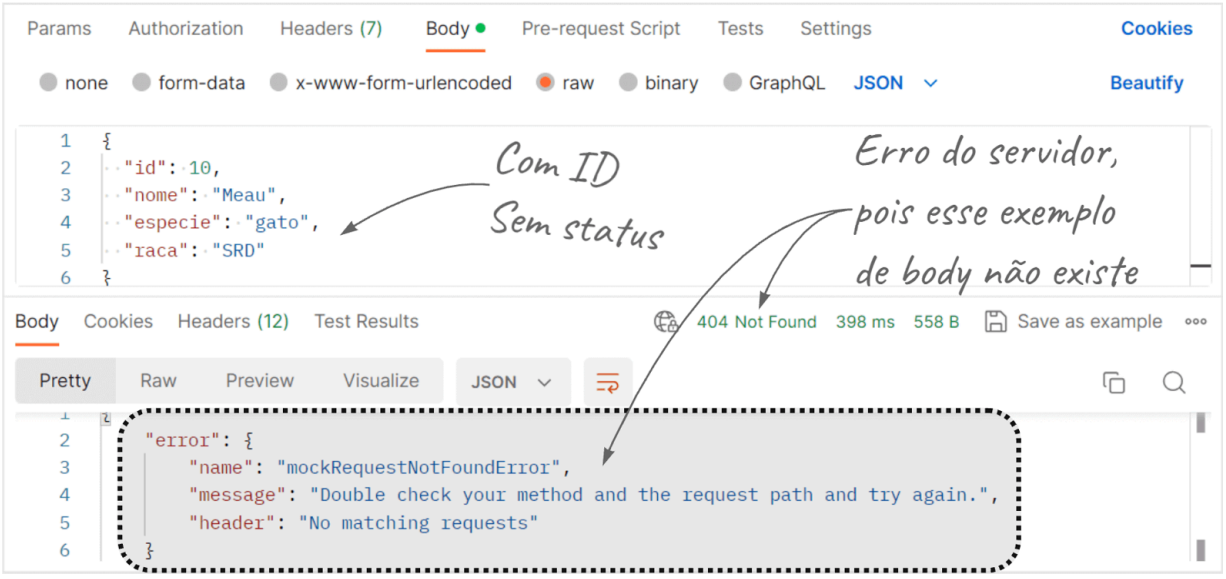


Figura 18.20: Resposta não encontrada.

As demais requisições previstas no *wireframe* têm a mesma estrutura e são criadas da mesma forma. Então, vamos conhecer a única diferente da anterior, a busca por um pet específico. Quando se quer consultar um registro específico, a requisição é do tipo `GET` e o ID do pet a ser consultado vai na URL, não no *body* da requisição. Para criá-la, posicione o mouse sobre o nome da *collection* — "Protótipo Petflix" — e clique em "*Add request*" no menu que é exibido ao movimentar o mouse sobre os três pontos.

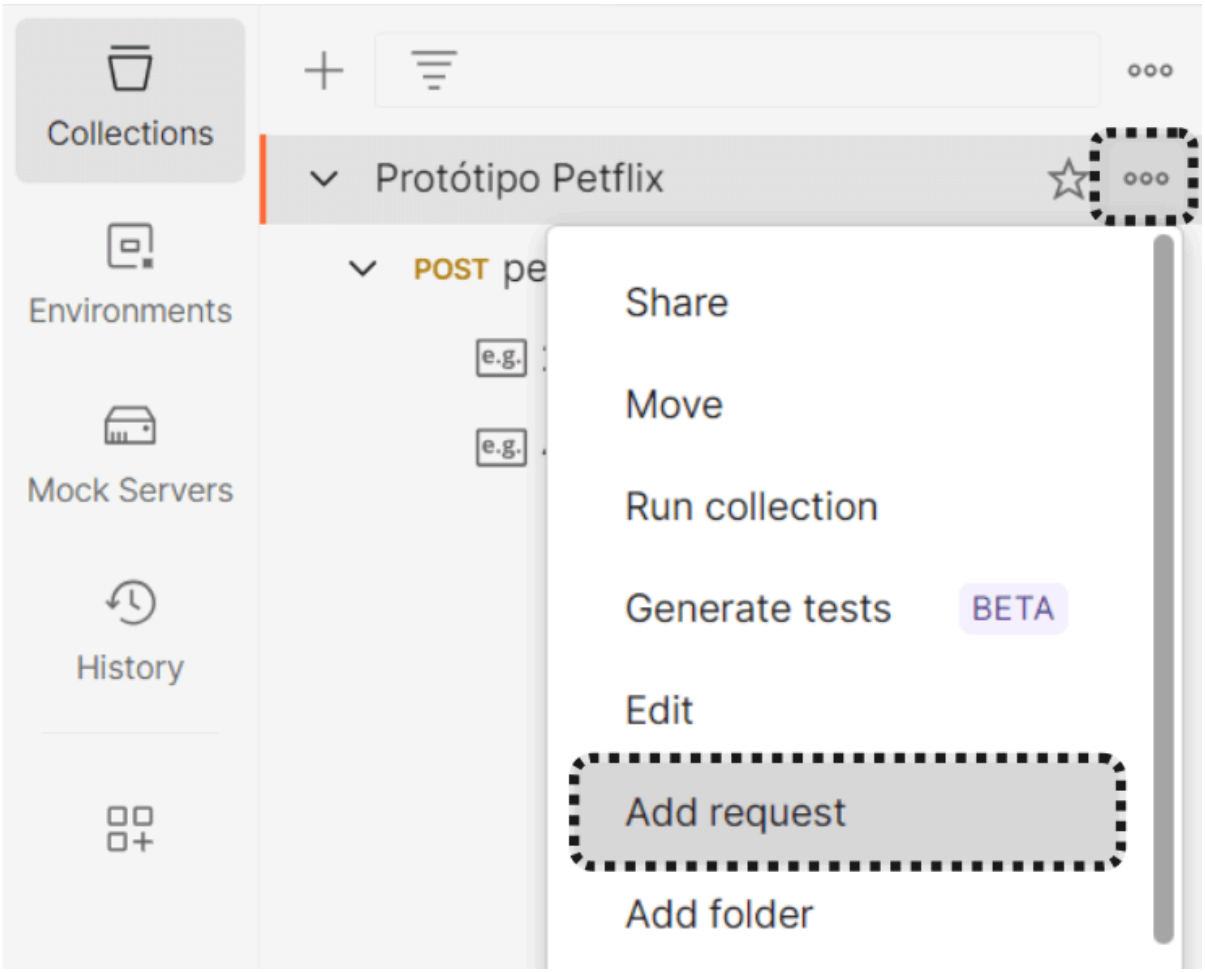


Figura 18.21: Criando nova requisição.

Abrirá uma aba. Nomeie a requisição como "Lista pets". O campo da URL vem em branco, então copie da requisição `POST` e cole no respectivo campo, como mostra a imagem, e clique em "*Save*".

Figura 18.22: Criando nova requisição.

Agora, você deve seguir o mesmo processo da requisição anterior para adicionar o exemplo — você pode duplicar o exemplo, se

preferir. Na aba que se abrirá, insira `/10` ao final da URL, como no print a seguir. Dessa vez, como a requisição não tem *body*, vá direto para a resposta, inserindo os dados do pet de ID 10 e o status `200 OK` . Salve clicando no botão "*Save*".

Figura 18.23: Criando exemplo para sucesso.

Faça um teste mudando o ID da URL por 11. Você receberá a mensagem de erro do Postman, porque o exemplo com 11 não existe. Agora, coloque novamente o 10 na URL e você verá os dados do pet na resposta. Lembre-se de que os exemplos são estáticos, então, se você quiser disponibilizar mais IDs na URL, precisará criá-los.

E assim, você pode prosseguir com todos os exemplos que você queira criar ou editar. Depois de criadas todas as requisições contempladas no *wireframe*, chegamos a algo como o próximo *print* e para acessar a documentação que esse protótipo gerou, clique em "Protótipo Petflix" e nos três pontos selecione "*View documentation*".

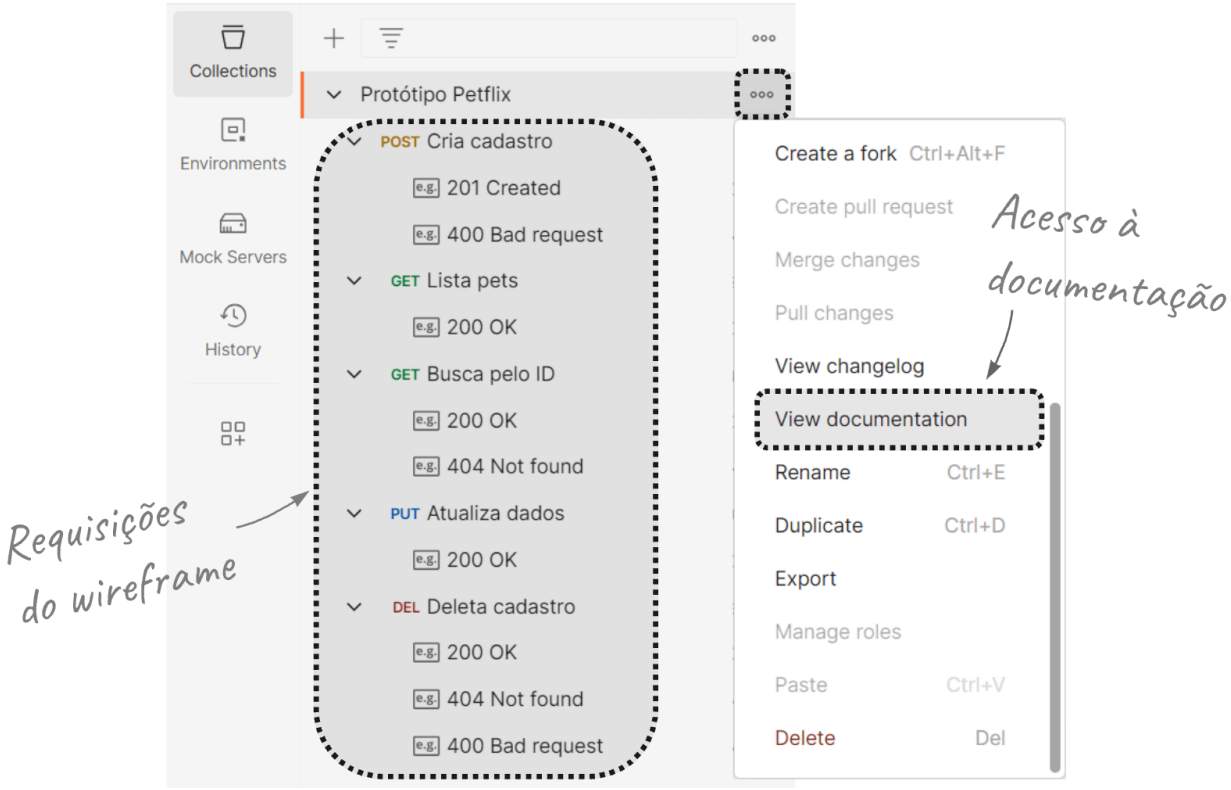


Figura 18.24: Abrindo a documentação.

Documentação

A documentação é a fonte para o teste de usabilidade. Ela permite que os devs usuários saibam quais *endpoints* estão disponíveis e como interagir com eles. O Postman criou automaticamente uma para nós, com *collection*, detalhes das requisições e até exemplo de código em diferentes linguagens de programação. É possível adicionar descrição, formatar o texto, inserir links, imagens, vídeos e tabelas.

Figura 18.25: Documentação editada.

Quando a documentação estiver pronta, você precisará publicar para que os devs possam testar. Publique clicando em "*Publish*" no topo da página da documentação, onde aponta a figura. Assim, a documentação fica disponível para qualquer pessoa que tenha o link e, à medida que a *collection* é atualizada, a documentação atualiza automaticamente.

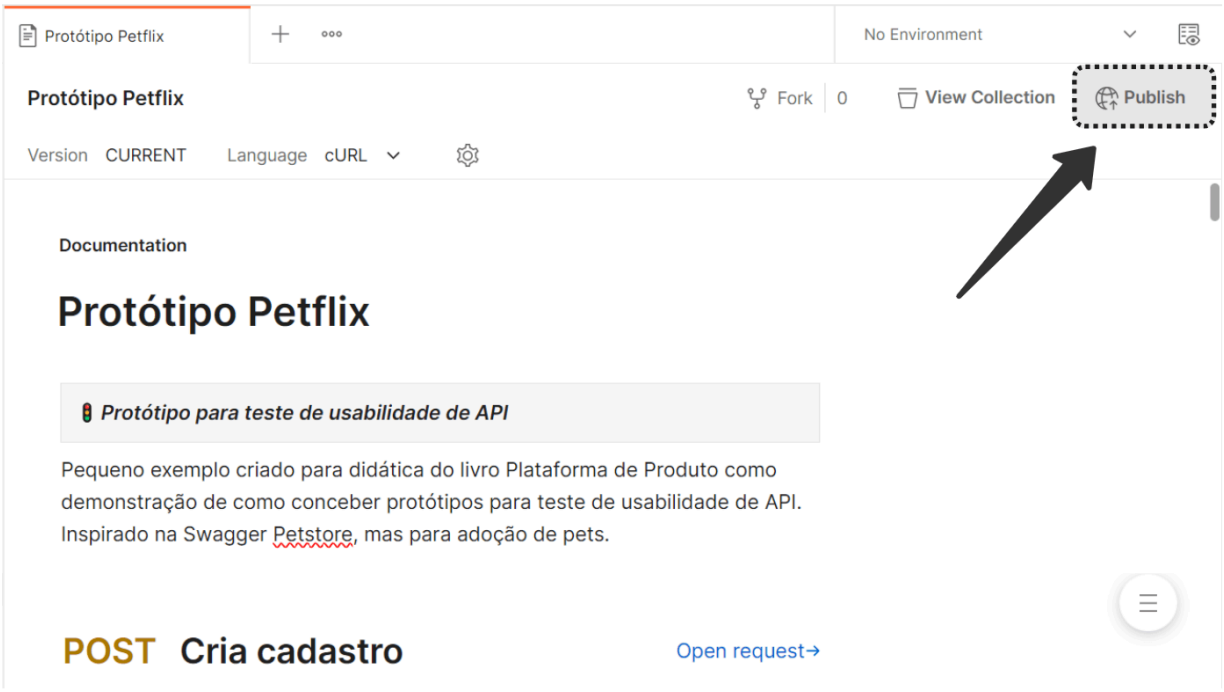


Figura 18.26: Publicando a documentação.

Isto feito, abrirão as opções de configuração. Você consegue escolher um visual com tema "*light*" (claro) ou "*dark*" (escuro), inserir logo, entre outras coisas. O resultado pode ser conferido na imagem que segue. Este é o material que será colocado à prova no teste de usabilidade.

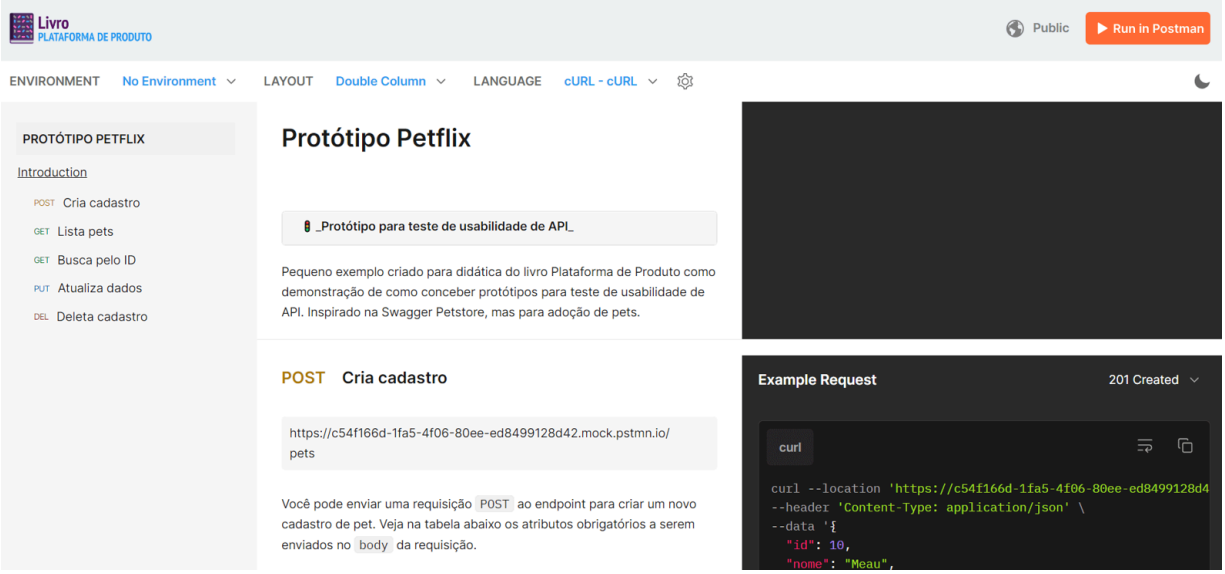


Figura 18.27: Documentação.

Essa documentação está disponível em https://bit.ly/postman-petflix. Clique no botão "*Run in Postman*" no topo da página, à direita, para acessar e testar os exemplos que criamos e os outros que estão no *wireframe*.

18.5 Teste de usabilidade

A essa altura, você já deve ter uma boa noção do que você pretende descobrir com o teste, mas ainda é necessário determinar algumas coisas e documentá-las em um planejamento com o roteiro do teste.

Planejamento e roteiro

A primeira coisa é decidir o objetivo: o que é necessário descobrir para avançar para o desenvolvimento com algum nível de segurança de que a ideia será bem-sucedida. Então, em um documento de texto, podemos colocar algo como:

OBJETIVO

O teste de usabilidade visa responder às seguintes questões:

- Os devs estão familiarizados com a tecnologia?
- É possível integrar apenas consultando a documentação?
- Quanto tempo leva para descobrirem o código e testá-lo?
- A API possui os atributos que os times precisam para disponibilizar as funcionalidades aos usuários finais?
- A API está seguindo as convenções de nomenclatura do setor de animais de estimação?

Para obter as respostas, o teste de usabilidade será conduzido remotamente — de modo síncrono — com os devs front-end do time A, devs Android e iOS do time B e devs back-end do time C.

Perceba que já foi definido como e quem são as pessoas (times) que vão trazer a resposta às perguntas. Se já existe uma versão da API, inclua quem já a utiliza e também pessoas de potenciais times que possam vir a utilizá-la.

E como será a sessão com essas pessoas? Vamos começar o roteiro com as **questões de introdução** para saber se a pessoa está familiarizada com REST, pois ela pode levar mais tempo para cumprir as tarefas e isso não quer dizer que a API e a documentação estão ruins.

- Você já fez integração com alguma API (REST, SOAP, GraphQL etc.)?
- Qual API você mais gostou? Por quê? — Essa pergunta pode levar a estudos de *benchmark*.

Agora, vamos elaborar as **tarefas e as questões relacionadas a elas**, de modo que nos ajude a verificar se a pessoa que está testando entendeu o que aconteceu na interação, se foi intuitivo e o

que ela mudaria. A experiência começa por uma visão geral da documentação e depois é passada a tarefa.

- Imagine que você precisa criar o cadastro de um pet. Crie um pet com estes dados: Nome: Meau; Espécie: gato; Raça: SRD; Status: adotado. — Essa tarefa mostra se está fácil de encontrar o conteúdo e se está claro; o que precisa ser feito para entender o que cada recurso representa e como deve ser requisitado.
- Por que você acha que recebeu um erro? O que você faria para corrigir? — A tarefa vai retornar um erro e essa pergunta é para saber se a mensagem de erro está clara e se o modo de correção fornecido ajuda de fato.
- Esse pet que você está tentando criar tem o ID 10. Faça a requisição novamente, adicionando esse atributo. O que é possível entender dessa resposta? — Dessa vez, o retorno será de sucesso, e essa pergunta serve para responder se o status de retorno e o conteúdo da resposta está de acordo com a expectativa do dev.

Uma vez elencadas todas as questões, você pode **concluir o teste**, com as perguntas de fechamento ou pedir para respondê-las em um formulário.

- A nomenclatura dos recursos e atributos estão de acordo com o significado? O que faltou ser representado? — Esta questão responde se os nomes dos recursos são descritivos, e se há algo que o time cliente precisa e não foi contemplado.
- Quais foram os maiores problemas ou dificuldades que você encontrou para cumprir as tarefas?
- O que você faria para melhorar a documentação?
- O que você melhoraria no design da API?

É provável que você precise de ajuda das pessoas de engenharia do seu time para elaborar as questões e as tarefas, porque elas têm mais noção do que pode causar atrito e elas mesmas têm dúvidas sobre o design. Você pode se valer da *Matriz CSD* para isso.

Durante o teste de usabilidade com os devs, lembre-se de mensurar o tempo que a pessoa leva para encontrar o que ela precisa na documentação para completar a tarefa e se recuperar de erros. Anote se ela conseguiu resolver sozinha e as dúvidas que você teve que responder a ela. Repare se ela teve de consultar outras fontes, como StackOverflow, e o tempo que levou procurando por informações adicionais. Isso servirá de insumos para complementar a documentação depois.

Neste momento, você dever estar pensando que tudo pareceu simples demais para ser verdade. Você tem razão. A realidade é mais complicada.

18.6 Prototipagem avançada

O comum é que as empresas adotem a estratégia de plataforma quando o contexto está, no mínimo, mais complicado — muitas das vezes, complexo. Dificilmente você verá uma API com apenas um recurso que contém meia dúzia de atributos a serem manipulados. A Order Plaform do iFood, por exemplo, possui centenas de atributos com vários níveis de hierarquia, dois tipos de APIs — síncrona e assíncrona —, *fallbacks* em inúmeros pontos da arquitetura e, para completar o combo, a gigantesca plataforma de dados.

Mantendo um panorama simplista, vejamos uma forma mais avançada — para nós, PMs não técnicos — de prototipagem definindo "*o contrato por meio de uma linguagem que expressa a especificação da API*", o item 1 citado no começo da seção *Prototipagem*. Confuso? Então, é hora de se orientar.

Definição da API

Você aprendeu uma maneira rápida de validar as hipóteses relacionadas ao design da API, com dados simulados e estáticos. Mas nem sempre essa será a forma ideal. Algumas vezes, as interações precisarão ser mais realistas, permitindo que os usuários cometam erros verdadeiros. Então, em vez de elaborar um cenário específico, com comportamento estático como fizemos com o exemplo do ID como atributo obrigatório, a resposta deverá contemplar cenários de forma mais dinâmica.

Nesta situação, o *wireframe* é formalizado em uma **definição de API** — comumente confundida com *especificação de API* —, que é convertida em um **contrato compreensível tanto por pessoas quanto por máquinas**. Os padrões de definição de API mais comuns são OpenAPI Specification (OAS), RAML e API Blueprint.

Lembra da documentação da Swagger Petstore apresentada no capítulo anterior? A imagem adiante do Swagger Editor mostra aquela documentação, à direita. Ela foi gerada a partir do código em linguagem YAML — definição do contrato — que aparece à esquerda, que segue o padrão de especificação OpenAPI versão 3.0. Nesse momento, ainda não é a implementação, continua sendo apenas um protótipo que funciona com um *mock server*.

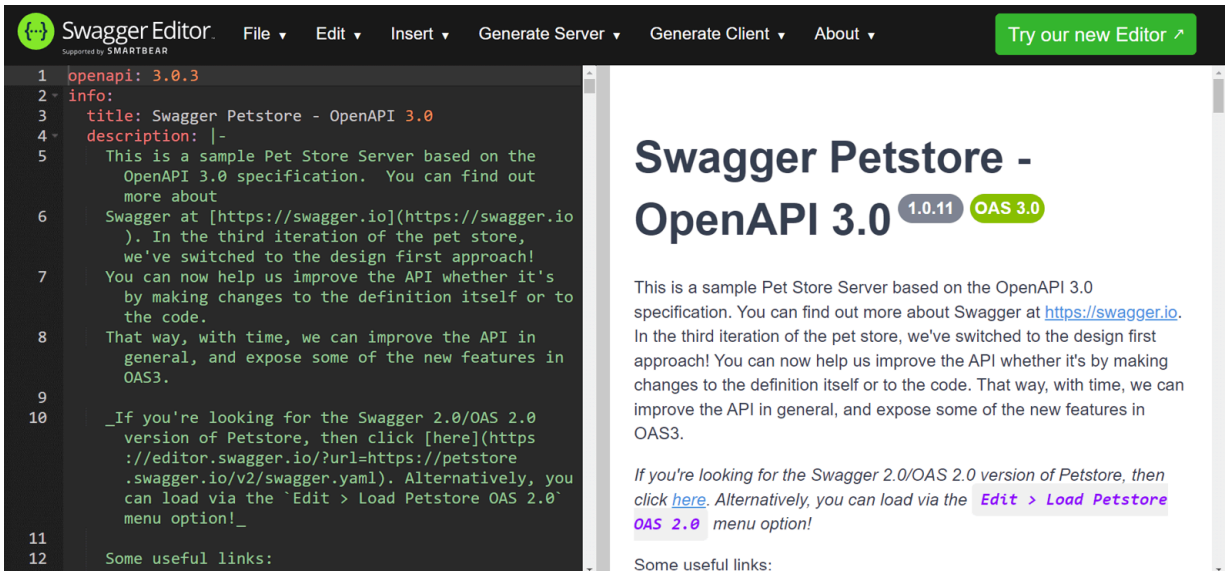


Figura 18.28: Swagger Editor

O Swagger Editor carrega a Swagger Petstore aberta ao público como exemplo e pode ser usada como template, possível de alterar e de fazer download. Conforme se altera o código, a documentação é atualizada. Veja na próxima imagem que, ao alterar o `title`, o título da documentação foi atualizado em tempo real de "Swagger Petstore - OpenAPI 3.0" para "API Petflix".

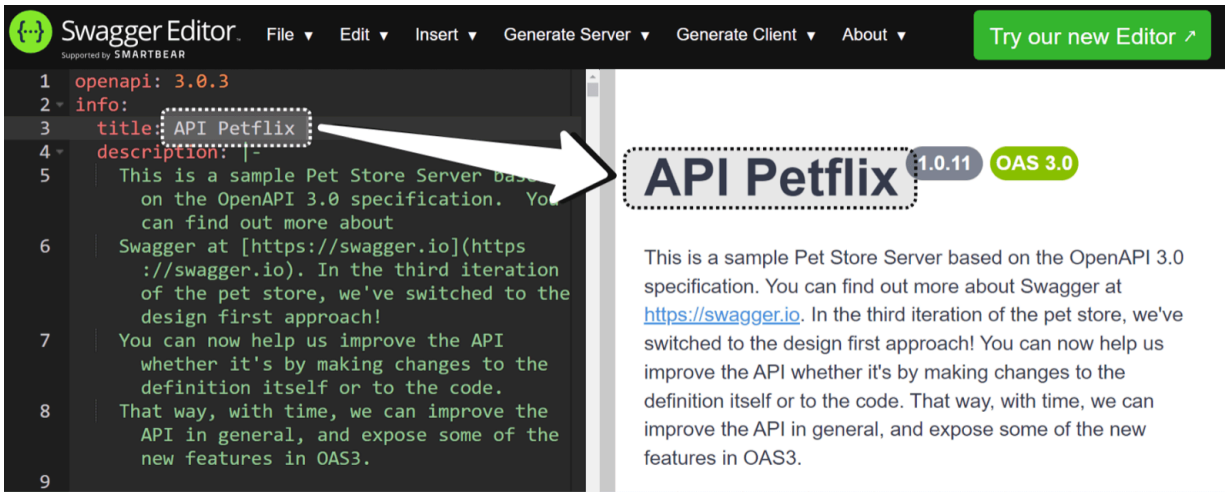


Figura 18.29: Atualização automática em tempo real

Sem aprofundar demais no *techniquês*, no *print* adiante podemos ter uma noção da diferença entre a definição da API se comparado ao que fizemos no Postman. Onde se vê `$ref: '#/components/schemas/Pets'`, significa que são links, exatamente como conhecemos. Neste caso, o link é para a estrutura de dados dos atributos de `pets`.

Figura 18.30: Atualização automática em tempo real

Supondo que queiramos adicionar um atributo a `pets` no protótipo que fizemos no Postman, por serem estáticos, todos exemplos em que se queira o novo atributo devem ser alterados para manter a uniformidade. Quando se define o contrato por meio de uma linguagem de definição de API, basta mudar na origem e todos os locais com o link carregarão o conteúdo da origem. Para APIs grandes ou complexas, essa alternativa pode diminuir esforço comparado ao protótipo estático.

Na verdade, a definição da API é uma atividade obrigatória para que ela possa ser usada, pois é versão funcional da interface. Isto é, o protótipo rápido não exclui a necessidade do "protótipo avançado". No entanto, se a validação rápida é suficiente, a definição final pode ser executada como parte da implementação.

E assim, com a API devidamente testada e validada, é hora dar de vida a ela. A fase de *Discovery* chega ao fim, mas o trabalho está

longe de acabar. Na Parte 4 e última, passaremos pelo que vem depois disso — Delivery, Lançamento e Monitoramento —, para que a API funcione, evolua e atraia o interesse dos clientes internos.

PARTE 4 — Depois do discovery

CAPÍTULO 19
Em busca de...

Sabemos que há duas perspectivas diretamente relacionadas às fases pelas quais um produto passa: o ciclo de vida do produto (introdução, crescimento, maturidade e declínio) e o ciclo de vida de desenvolvimento dele (*discovery*, *delivery*, lançamento e monitoramento) — assunto abordado no capítulo *Gestão do ciclo de vida*, primeira parte deste livro. Os capítulos da Parte 3 foram dedicados ao *discovery*, fase em que se busca eficácia.

E agora, vejamos em linhas gerais as demais fases que se concentram na busca de:

- Eficiência no *delivery*
- Adoção no lançamento
- Retenção pós-adoção

19.1 Em busca de eficiência no delivery

A implementação do back-end e tecnologias que tornam o que foi idealizado em uma plataforma funcional envolve as mesmas etapas de produtos *customer-facing*: desenvolvimento, testes e deploy.

Esta é uma fase fortemente associada à busca por eficiência em termos de produtividade e, para alcançá-la, muitas empresas aderiram a alguma metodologia ágil como Scrum ou Kanban.

Se você trabalha com produto de software há alguns anos já percebeu que a adoção e conformidade ao método não garante produtividade. Como há um acervo gigantesco sobre as metodologias, inclusive em português, não veremos esse assunto por aqui. Vamos nos restringir à seguinte reflexão: o que é eficiência no contexto da agilidade.

Agilidade e eficiência

No capítulo de introdução à Parte 3, *Em busca de eficácia*, foi comentado que há empresas que aplicam alguma metodologia ágil por motivo equivocado: produzir a maior quantidade de coisas no menor tempo possível. Esse é o tipo de eficiência consagrada pela indústria de manufatura, adequado a trabalhos e processos repetitivos, que produzem o mesmo produto sem alterar sua eficácia. Não há riscos em se produzir mais do mesmo (e da mesma forma), portanto é coerente colocar máquinas para trabalhar na capacidade máxima.

Desenvolvimento de produtos é um trabalho criativo. Neste contexto, a quantidade do que se produz deixa de ser importante. O que importa é o quão bem a criação endereça o problema, a eficácia. Ninguém reclama quando parte dos itens de um escopo ficou fora da entrega, mas o que foi entregue resolve bem o problema. E quando acontece o contrário? Reclamações não param de chegar quando o problema não é resolvido, mesmo que se tenha entregado todos os itens do escopo.

A quantidade de entregas não é o que deixa stakeholders felizes, por isso é incoerente medir eficiência de desenvolvimento de produtos seguindo o modelo tradicional. Claro que queremos eficiência, mas quando isso se confunde com quantidade de coisas que se faz, o que se obtém? Pressa.

Quando se tem pressa, usa-se a capacidade máxima do time para implementar tudo o que está no backlog e o quanto antes. Nesse caso, o objetivo é ver todos os itens na coluna *done* e a performance

do time é mensurada pelo quanto se queima de escopo (*output*) e não pelo *problem-solution fit* (*outcome*).

Diante disso, o time deixa de trabalhar na coisa certa, na hora certa, o que induz ao risco de entregar algo que não resolve o problema. Quando isso acontece, os stakeholders cobram pela resolução de uma dor que ainda sofrem e isso provoca a entrada de mais itens no backlog.

Quanto mais o backlog cresce, maior é a pressão para que devs produzam mais, "*levando-os a pularem etapas importantes, como testes automatizados, refatoração, reorganização do código, otimização de performance e documentação*" (VALE, 2020). É um comportamento que gera sequelas: bugs. E bugs provocam a entrada incontrolável de demandas.

Percebe que ao mirar na queima de escopo se obtém o oposto? O backlog aumenta em vez de diminuir. "*Isso sugere que quando as organizações se concentram demais na utilização eficiente dos recursos — a forma tradicional e mais comum de eficiência — isso tende a levar a um aumento na quantidade de trabalho a se fazer. Consequentemente, quanto mais as organizações tentam ser eficientes (estando ocupadas), mais ineficientes elas se tornarão (cuidando do trabalho que não agrega valor, mas necessário)*", explica os autores do livro *This is Lean: Resolving the Efficiency Paradox* (MODIG; ÅHLSTRÖM, 2012, apud VALE, 2020).

Diante de inúmeras interrupções no fluxo de trabalho e imprevisibilidade, é impossível cumprir o que foi planejado. Plataformas ainda têm um agravante. Como um tsunami que arrasta o que estiver no caminho, bugs propagam-se em todos os apps que derivam da plataforma e atingem todos os clientes finais que usam esses apps, o que pode levar ao prejuízo de alguns milhões de reais (ou dólares) para o negócio. Tudo o que é construído no nível de plataformas exige rigor de qualidade.

Qualidade e pressa não combinam, tampouco qualidade e eficiência em termos de produtividade como da indústria. Quando se trata do trabalho do conhecimento (criativo), o volume de coisas que cada pessoa produz, ou o quanto elas escrevem de código, é irrelevante.

O desenvolvimento de produtos se torna lento devido a interrupções, bloqueios, filas e espera entre etapas (*hand-offs*). Essas coisas roubam o tempo disponível que o time tem para trabalhar. Independente de haver scrum master ou agilista no time, nosso papel como PMs é a busca por um modelo de eficiência que dê suporte à eficácia, a **eficiência de fluxo** — gestão acerca do que impede o fluxo de fluir, de modo a evitar que os itens parem no meio do processo.

Para começar, é preciso trazer à consciência que, com ou sem *discovery*, o backlog é cheio de hipóteses. Ainda há incertezas (inclusive ocultas) que vão se revelar na fase de *delivery*.

Backlog é cheio de hipóteses

Em algum tipo de cerimônia equivalente a uma *planning* do scrum, você e o time definem coisas que precisam ser feitas para construir a solução. Essas coisas viram itens (*cards* com tarefas) no backlog. Conforme os itens avançam, aparecem obstáculos que não foram previstos, como problemas no código ou incompatibilidade entre tecnologias e módulos — você deve ter passado por isso centenas de vezes.

Algumas coisas *sabemos que não sabemos*. Essas são fáceis de verbalizar e podemos escolher prosseguir apesar dos riscos, ou não. Mas há tantas outras que nem sabemos que não sabemos, o fenômeno ***unknown unknowns***. Inesperadas, é impossível prevê-las, portanto, é impossível planejá-las.

Nós, do time da Order Platform, elencamos o que precisaria ser feito para determinada entrega de valor em um *user story map*. O desenvolvimento começou e logo nos primeiros dias percebi que

havia vários (vários mesmo!) novos *cards* no backlog. Você pode pensar que o mapeamento não foi bem feito — eu mesma cheguei a pensar que esses itens deveriam ter aparecido na sessão de *story mapping* — porém, no momento em que se define o que precisa ser feito, parte da consequência está oculta.

Ainda dentro desse contexto, sabíamos que era necessária alguma forma de autenticação, então iríamos criá-la. Durante o desenvolvimento descobriu-se que a tecnologia escolhida para essa implementação contemplava uma forma de identificar os sistemas consumidores da plataforma, suficiente para o que precisávamos. Sem medo de ser feliz, os itens referentes à criação da autenticação foram descartados.

Esses exemplos ilustram que o mapeamento *sempre* será com base no conhecimento que se tem até o momento. As incertezas tornam-se conhecidas durante a construção e testes. Somente depois de conhecido é possível considerar o que pode ser feito e, a escolha do que fazer é fundamentada no que o time *acredita* que funcionará bem e resolverá o problema. Sendo crença, mais uma vez, caímos no campo das hipóteses.

Gestão do backlog

Muito se fala de hipóteses na fase de *discovery*, nunca na de *delivery*. A verdade é que os itens do *delivery* são hipóteses que serão verificadas ao longo do desenvolvimento ou somente depois de concluídas. Se é assim, precisamos minimizar o tempo entre o momento em que a hipótese da solução surge e sua validação. Como se consegue isso? Por meio da eficiência de fluxo mencionada anteriormente.

A primeira coisa a se considerar é que a entrada de itens no backlog não significa obrigação de serem executadas. O objetivo **não** é manter as pessoas ocupadas ou queimar escopo, levando todos os cards para *done* só porque está no backlog. Qual é a vantagem de fazer com eficiência algo que nem deveria ser feito?

Eficiência só tem valor quando o alvo é a eficácia — execução eficiente da coisa certa e na hora certa. Talvez isso signifique concluir todos os itens do backlog; talvez precise adicionar mais alguns. No entanto, ao puxar cada *card* deve-se questionar:

- Com base nas informações que se tem hoje, esse item ainda faz sentido?
- Essa tarefa é imprescindível para construir a solução que vai levar ao determinado resultado?

Se a resposta é "não" — o item deixou de fazer sentido ou não vai levar ao resultado esperado — simplesmente se não faz! Descarte sem dó o que não agrega valor. Se é "sim", a próxima pergunta é: precisa ser feita agora? "Não", então deixe-a para fazer no momento em que ela se tornar relevante. A hora certa será quando se mostrar impossível ou arriscado demais prosseguir sem ela. Então, se a resposta é "sim", essa é a hora certa de fazê-la.

Na prática, a entrada de itens no backlog é mais complicada do que isso. Apesar do planejamento, compromisso com OKRs (ou metas de negócios), chega um momento em que algum cliente interno lembra que depende da sua plataforma para algo que estão desenvolvendo; ou então alguma coisa inesperada acontece e outro time cliente precisa resolver e uma das ideias exige algo novo na sua plataforma. Supondo que ambas sejam problemas genuínos e faz sentido incorporar na sua plataforma, o que priorizar?

Priorização de demandas

Assim como qualquer produto, plataformas são repletas de demandas não planejadas. Na gestão de produtos *customer-facing*, a priorização pode ser por meio de algum *framework*, como *Matriz de esforço x impacto*. É pouco provável que essa seja a ferramenta adequada para a sua plataforma, uma vez que ela deve suportar o negócio no longo prazo. Pequenos ganhos ou *quick-wins* podem levar a problemas grandes.

Supondo duas demandas de dois times clientes. Você discute as opções para ambos com seu time e, se as demandas concorrem no mesmo grau, o melhor trabalho que você pode fazer é atuar como pessoa facilitadora da conversa entre os stakeholders. Se você se comprometeu com alguma frente estratégica e essas demandas colocam em risco a entrega planejada, é o caso de envolver a alta gestão. Você expõe as demandas de ambos, o andamento das coisas com que o time já estava comprometido, as opções do que pode ser feito e deixe o *trade-off* nas mãos deles: o que sai para que a demanda não planejada possa entrar. Garanta que a troca que os stakeholders estão fazendo e as consequências disso fiquem explícitas. Mostre evidências sobre alguns pontos a que eles devem se atentar:

- **É remendo ou "gambiarra"?** Esse tipo de coisa em plataformas tem um efeito exponencialmente pior do que em produtos *customer-facing*. Traz fragilidades que refletem em tudo o que deriva da plataforma, além de ser difícil garantir alguns dos valores característicos de plataformas: performance e escalabilidade no longo prazo. Esses fatores podem inviabilizar todo o negócio em algum momento.
- **Quais as dependências?** Uma pequena mudança na plataforma pode exigir o trabalho de diversos times clientes para manter a empresa operante. Dependendo da quantidade de aplicações envolvidas, pode custar caro e demorar para que todos os times executem as mudanças, retardando a entrega de valor.
- **É uma implementação de plataforma?** Há demandas que as pessoas supõem que deveriam ser absorvidas pelas plataformas, mas são tão específicas a determinado contexto que não é coerente plataformizar. Idealmente, plataformizam-se *features* das quais mais de um time possa se beneficiar e que já estejam validadas. Plataformas são perfeitas para acelerar experimentos que usam o que já existe na nela, contudo não deveriam ser motivadores para se criar ou modificar coisas na plataforma. Além do investimento em tempo e esforço, é um

desperdício manter algo que ninguém usará se a hipótese for invalidada.
- **Podem esperar até que esteja pronto?** Para acelerar o desenvolvimento dos apps dos times clientes, é natural que complexidades sejam absorvidas pela plataforma. A vantagem é que ela é desenvolvida uma única vez; a desvantagem é que leva mais tempo para entrega de valor e provoca a sensação de que as coisas demoram demais — até pessoas da alta gestão de engenharia esperam que plataformas sejam desenvolvidas na mesma velocidade (ou cadência) que produtos *customer-facing*.

Na criação de uma nova plataforma ou de algo novo em uma plataforma existente, você vai se deparar com muita infraestrutura para construir e configurar. Até que o funcionamento da fundação fique redonda, serão necessários inúmeros ajustes até estabilizá-la. Depois disso, embora se tenha muita coisa pronta e funcionando, não traz valor para os times clientes, para os usuários finais e nem para o negócio.

Isso gera pressão sobre o time da plataforma, afinal tudo o que se espera de plataformas é que ajudem a acelerar o *time-to-market* dos apps/produtos. Há uma saída para diminuir a ansiedade dos stakeholders. Você saberá qual é, na seção *Em busca de retenção pós-adoção*. Antes, vamos olhar para o lançamento.

19.2 Em busca de adoção no lançamento

Lançamento de produtos abrange atividades como: comunicação interna, treinamento para o time de vendas e atendimento, material de ajuda para os usuários e informações para a área de marketing divulgar ao mercado-alvo. As pessoas de marketing elaboram a mensagem de posicionamento e promovem o lançamento por e-mails, posts em redes sociais, entre outros canais. Produto e marketing definem suas respectivas métricas para

acompanhamento. Essas atividades também se fazem presentes para lançamentos de plataformas voltadas para o público externo.

Plataformas internas parecem mais com lançamento de produto interno. É improvável que tenha alguém de marketing para ajudar na divulgação interna, mas certamente alguém terá de fazê-la. Esse alguém costuma ser a pessoa PM. No caso de plataformas, além de PMs, é comum que seja a pessoa engineering manager devido ao "techniquês". Vocês devem anunciar aos times, principalmente, se os times clientes tiverem que implementar alguma coisa.

Em se tratando da liberação para uso — *rollout* —, é de praxe que produtos iniciem com um grupo específico de *early adopters*, ou aleatório, e amplie gradualmente até que seja disponibilizado a 100% dos usuários. Produtos internos talvez não precisem desse cuidado; plataformas internas já são outra história. O lançamento de uma nova versão da plataforma pode ser tão complexo que o mais sensato é definir um plano de lançamento e migração, antes mesmo da implementação.

Comece pelo plano de migração

A versão em produção da Order Platform era robusta e impressionante nos quesitos resiliência e performance. Porém, processar o volume de pedidos estimado para os próximos anos colocava o crescimento do negócio em risco. Antes da minha chegada, o time tinha uma certeza: a nova plataforma precisava mudar de tecnologia.

Os primeiros meses foram dedicados ao *discovery* dessa tecnologia por meio de PoCs. Também foram avaliadas algumas soluções de mercado que ajudassem a acelerar o desenvolvimento de certas *features*. Depois, partimos para o MVP que contemplava a implementação da fundação — o funcionamento do *core* — sem a qual seria impossível a nova plataforma existir. O objetivo desse MVP era validar a performance para lidar com tráfego e transmissão de massa de dados colossais, pois a tecnologia era nova para o time.

Para isso, teria que passar 100% dos pedidos pela nova base estrutural (os mesmos processados pela plataforma vigente), mantendo a consistência de dados entre ambas. Portanto, pedidos alterados em produção deveriam refletir na nova. Desafios tão complicados de ciência da computação e engenharia, de execução minuciosa, que "só isso" levou quase quatro meses e nenhum valor foi entregue aos usuários e ao negócio.

Por outro lado, foi validado que o poder de processamento da tecnologia e a arquitetura escalável (quase ao infinito e além) não colocaria o crescimento do iFood em risco no longo prazo — o que motivou a criação de uma nova versão, mas valor a ser percebido somente no futuro um pouco distante.

A nova versão da Order Platform deveria fornecer, no mínimo, a mesma robustez e *features* que a versão vigente, pois ela seria descontinuada. Todos os times usuários teriam que migrar para a nova plataforma em algum momento. Isso significa que precisava estar tudo pronto e funcionando para os times migrarem?

É completamente inviável e imprudente envolver 32 times na migração de aproximadamente 150 aplicações somente depois de tudo pronto. Além disso, esperar a migração de todos os times levaria meses, talvez mais de um ano, para a plataforma gerar valor. Haveria alguma forma de liberar o uso e gerar valor gradualmente? Quais times deveriam ser os primeiros a migrar? O deveria ser priorizado após o *core*?

Nem todas as aplicações usam tudo o que a Order Platform fornece, mas, por ser uma plataforma central, ela alimenta outras plataformas que alimentam os apps. Uma única operação depende de algumas plataformas, como quando uma pessoa faz um pedido de refeição pelo app: (1) ela escolhe um restaurante e seleciona o que vai consumir; (2) depois, ela confere os itens e os preços no checkout, (3) executa o pagamento e, nos bastidores, (4) o pedido é

criado dentro do iFood pela Order Platform. Só nessa parte do processo, há pelo menos três plataformas envolvidas e todas elas se comunicam com a Order Platform. Este é um exemplo de que é obrigatório que essas três estejam integradas à Order Platform para viabilizar *somente* a primeira parte da jornada de compra de refeição.

Em um contexto como este, você poderia definir um **plano de migração conservador**, que evite a queda de alguma métrica importante para o negócio. Por exemplo, viabilizar a operação que possui o menor número de pedidos e que também dependa da menor quantidade de serviços previamente integrados à plataforma. Desse modo, quando surgirem problemas — e sempre surgirão — a perda de pedidos será mínima e haverá tempo para os times dos serviços integrados entenderem cada incidente e atuarem. Conforme mais aplicações migram, mais operações são destravadas. Quando a operação de maior volume de pedidos for viabilizada, em tese, a maior parte dos problemas está resolvida.

Outra abordagem seria um **plano de risco moderado**, priorizando uma *feature* que já demonstra necessidades latentes, aquela cuja versão em produção não atende bem aos variados casos de uso e impede que os times dos apps ofereçam melhor experiência ao usuário. Por exemplo, uma limitação ou atrito na jornada dos usuários finais, algo que o usuário gostaria de fazer no app, mas não consegue e causa transtornos à operação. Neste caso, a intenção é viabilizar uma parte da jornada do usuário final adicionando capacidades na nova versão da *feature*, sem tanto foco aos riscos relacionados à migração. Não quer dizer desconsiderá-los; quer dizer que o objetivo não é mitigar riscos da migração, o objetivo é liberar novas capacidades para os times clientes. Por essa lógica, as migrações ocorrem conforme as *features* são disponibilizadas.

Dependendo do momento da empresa, a migração pode seguir um **plano mais arrojado**, como viabilizar a operação de uma unidade de negócio ou produto que ainda não usa a plataforma. Nesta situação, a primeira coisa a ser feita é avaliar quais são os potenciais

produtos/apps que possuem solução própria para o que sua plataforma fornece. Depois, entender com os respectivos times como funciona a solução deles, as limitações, o que precisa ser melhorado e comparar com a promessa da sua plataforma. É importante saber também a tecnologia que usam, se a sua é uma boa opção ao contexto em questão e se estão dispostos a adotar a sua solução. Nessas conversas é a hora de vender a sua plataforma, deixar os times desejosos pelo que você está construindo. Além disso, é preciso um estudo cuidadoso de quais serviços essa operação dependerá, que juntos habilitam determinada operação, bem como o que o novo time cliente precisará adaptar no produto dele para se adequar a todos os serviços envolvidos.

Apesar de mais arriscado – exige que o novo cliente adapte-se à sua plataforma e às demais que viabilizam a operação –, um plano voltado para aquisição de um novo cliente é o ideal para validar se a plataforma é genérica e flexível para atender a negócios de naturezas diferentes.

A promessa da Order Platform é processar pedidos de (quase) qualquer natureza. Claro que é impossível prever o que a empresa vai passar a fornecer no longo prazo, no entanto, a plataforma deveria ser genérica e flexível o suficiente para suportar tipos de pedidos dos mais variados e, se não fosse possível, deveria ser extensível a novos formatos. Sem dúvida, o maior risco de valor seria não conseguir processar pedidos além dos já conhecidos. Por isso, optamos pelo plano arrojado, adquirir um time cliente cujo tipo de pedido ainda não era contemplado, nem mesmo previsto, na nova plataforma. Isso exigiu inúmeras adaptações na plataforma, nos serviços que viabilizam tal operação e também nas aplicações do time cliente.

Esses são apenas alguns exemplos de como você poderia definir um plano de migração. Quando bem-organizado, o plano traz visibilidade em uma sequência priorizada do que deve ser feito para liberar as capacidades e as migrações atreladas a elas. O sucesso disso depende da disponibilidade dos times clientes. Por isso é

imprescindível que eles saibam sobre a migração com bastante antecedência e gestão de programa (**program management**) é ideal para apoiar iniciativas grandes e complexas como essa.

Program management para gestão da complexidade

A migração de várias aplicações para uma nova tecnologia não é um processo trivial. Isso representa trabalho extra para os times clientes. Como você se sente quando uma demanda assim aparece do nada e com prazo curto para seu time implementar? O mínimo é que tivessem te avisado muito antes, né? Um caso como o da Order Platform precisa de um plano de migração bem definido e que seja comunicado com meses de antecedência. Se bem orquestrada, a migração de tantas aplicações envolvendo tantos times leva meses, imagine mal orquestrada. É aí que a gestão de programa faz uma baita diferença.

Nem toda empresa possui program managers — ou possui sob outra nomenclatura. Se você não ouviu falar, **programa** é uma iniciativa que contém vários projetos que contribuem na concretização de uma estratégia mais ampla do negócio. **Gestão de programa** (*program management*) é a coordenação dos inúmeros ***deliveries* interdependentes**. Inclui a visibilidade e o monitoramento do progresso dos projetos, além da comunicação de tudo o que possa afetar o progresso ou resultado do programa: atraso no cronograma, mudança de requisitos e/ou escopo e a descoberta de stakeholders, por exemplo.

Program managers não são membros dos times, por isso, não conhecem cada plataforma e app no detalhe, tampouco todos os times usuários. Nós, como PMs, é quem temos de fornecer as informações e a pessoa program manager nos ajuda na comunicação com os times envolvidos e com a alta gestão, antes mesmo de o programa iniciar.

Voltando à Order Platafom, sabendo quais times seriam os primeiros a migrar, houve várias conversas até a comunicação se formalizar.

Isso feito, esses times enumeraram o que seria feito e quando. Depois que o programa é iniciado, PMs e/ou engineering managers de cada frente fornecem atualizações periódicas à pessoa program manager, que sintetiza as informações em uma visão estratégico-tático de cada projeto e avalia como o programa está progredindo como um todo. Isso promove alinhamento entre stakeholders e fornece alguma noção de quando algo pode acabar para que outra possa começar em outro time.

Gestão de programa faz diferença considerável quando a comunicação e o acompanhamento do *status report* de tantos times é complexa e trabalhosa demais para ficar na responsabilidade da pessoa PM da plataforma. Essa operação é difícil e, se você não quiser tornar o processo ainda mais complicado, tenha tudo o necessário pronto para as migrações. É responsabilidade do time da plataforma tornar as migrações o mais suave possível para os times.

O canal para adoção

Sem algumas informações básicas é impossível que os devs usuários consigam utilizar a sua API — você constatou isso no capítulo *Como PMs conseguem testar APIs*. Então, antes que se inicie as migrações ou aquisições de times clientes é imprescindível que a **documentação de referência da API** (*API reference*) — como da Swagger Petstore — esteja pronta e disponível para consumo.

Às vezes, a documentação de referência é insuficiente. É preciso apoio de uma **documentação conceitual** (*conceptual documentation*). Muitas vezes chamada de **guia do usuário** (*user guide*), este tipo de documentação esclarece para que a API serve, como começar a usar, o que é possível fazer com ela, os casos de uso, as limitações e os problemas comuns que podem ocorrer durante a implementação.

Essas documentações compõem o conteúdo do **portal do desenvolvedor** (*developer portal* ou *dev portal*) — citado e exemplificado no capítulo *Segmento de clientes e canais*. Você

encontrará portais em que ambos os tipos de documentação são separados e outros em que o conteúdo é mesclado em uma só, não há uma regra. A estrutura e o conteúdo dependem do que precisa ser explicado e de como ficará fácil de entender.

No capítulo *Segmento de clientes e canais*, na seção *Canais de entrega de valor*, vimos que o portal do desenvolvedor costuma ser o canal principal, onde os times clientes passam do conhecimento de que a plataforma existe (*awareness*) para as etapas seguintes do funil apresentadas no capítulo *Métricas*, aquisição e ativação. O *dev portal* fornece aos usuários da API uma maneira mais conveniente de aprender e interagir com ela, mas desenvolvê-lo pode tomar um tempo que é mais bem aplicado na evolução da plataforma. No capítulo *Gestão do ciclo de vida*, vimos que uma das alternativas é utilizar a ferramenta para criação de portais que os gerenciadores de API fornecem. Esses sistemas possuem funcionalidades prontas que proporcionam uma experiência *self-service* completa aos devs usuários: acesso seguro, autonomia para registrar os apps deles, selecionar as APIs e os níveis de serviço, além de monitoramento de uso. Mas têm custo extra, na maioria das vezes.

Se custo é um problema, saiba que tem opções gratuitas e *open source*, como o Backstage (https://backstage.io/) criado pela Spotify. Pode ser que você não precise de um portal do desenvolvedor (no sentido literal, com todas as funcionalidades) para providenciar o que os times clientes precisam. O que você precisa é disponibilizar um canal que proporcione autonomia aos devs usuários, torne o processo de adoção *self-service* e, dessa forma, evite que eles dependam do seu time. Muitas vezes, basta um canal que seja o ponto de partida da jornada com links para as documentações.

Se a API for tão simples a ponto de nem precisar de muitas explicações, talvez a medida certa seja disponibilizar apenas a referência da API — como aqueles do Postman e do Swagger Editor explorados no capítulo anterior. No entanto, ainda é preciso algum nível de segurança que identifique as aplicações consumidoras das suas APIs e, assim, proteja a sua plataforma de uso não autorizado.

A seção *Aquisição* do capítulo *Métricas* apresenta uma ideia simples para cadastro e obtenção de credenciais para autenticação (*API Key*).

Falta de documentação é péssimo, mas ter uma e deixá-la desatualizada é igualmente ruim. Lembre-se de revisitar e atualizar a documentação antes do deploy de alterações ou de algo novo.

Por fim, uma informação relevante! Os devs usuários não precisam esperar até que o back-end esteja completamente desenvolvido para consumir a API. Quando o time da plataforma segue o método ***design first*** (design primeiro), devs usuários podem usar a documentação de referência de API para integrar, enquanto o time da plataforma implementa o back-end. Claro que não está em pleno funcionamento. Na prática, a interação se dá pelo *mock server*. Os dados não são reais e nada é salvo no banco, mas é útil para times clientes que querem testar como ficará a funcionalidade no app deles e dá até para fazer experimentos com os usuários finais. Só uma ressalva: é vantajoso desde que o contrato da API não sofra alterações, do contrário, os times clientes terão retrabalho — mais sobre isso na próxima seção.

Design first* versus *Code first

O mais comum é que times de plataforma sigam o método *code-first* (código primeiro). Se o seu time começa pela infraestrutura e desenvolvimento do back-end, para só então elaborar o design da API, você já está familiarizado(a) com este método.

O método *design first* (design primeiro) propõe o processo contrário, mais parecido com produto *customer-facing*. Primeiro o design e usabilidade (UI/UX) e, somente depois de testado com usuários, é que se desenvolve. É o processo do capítulo *Da ideação ao teste de usabilidade*. Inicia pela definição da API e depois do feedback dos devs usuários é que se elabora como tornar o contrato funcional.

Mas não se engane. Nem sempre *design first* é o melhor caminho. *Code-first* é ideal em situações em que o escopo é pequeno e quando cumprir prazo é mais importante do que garantir a longevidade de longo prazo.

19.3 Em busca de retenção pós-adoção

Você publicou suas APIs, divulgou internamente e os times adotaram sua solução — ou migraram para a nova versão. É hora de adentrar o último estágio do ciclo de vida de desenvolvimento do produto: monitoramento e mensuração dos resultados com objetivo de identificar melhorias para reter clientes conquistados, sem perder de vista futuras aquisições.

Quando os times não podem optar por soluções externas ou por desenvolver a própria solução, o risco de uma plataforma interna perder clientes é nulo. Mesmo assim, há risco de valor se o que a plataforma faz é muito discrepante do que ela deveria fazer ou

quando questões técnicas, como lentidão, afetam a experiência dos clientes finais e causam prejuízo financeiro ao negócio.

Umas das formas de se evitar *churn* (perda de clientes) é melhorar o que existe; outra forma é adicionar novas capacidades. Dados qualitativos, como feedback dos usuários, ajudam a entender se algo deve ser otimizado ou se há novas necessidades. O monitoramento contínuo e em tempo real de dados quantitativos ajudam a entender padrões de consumo das APIs, de performance e evidenciam oportunidades de melhoria.

Monitoramento e observabilidade

As vantagens da arquitetura de microsserviços têm seu custo: aumento da complexidade conforme mais e mais serviços interagem entre si. Em um contexto como esse, é dificílimo e trabalhoso identificar a causa dos problemas, por isso é crucial que a organização estabeleça uma estratégia de monitoramento e observabilidade.

Os termos *monitoramento* e *observabilidade* são comumente usados como sinônimos, mas não são. **Monitoramento** consiste na "*coleta, visualização e alerta acerca de métricas predefinidas para garantir que uma API atenda às expectativas específicas*" (POSTMAN, c2024; tradução livre).

Observabilidade (*observability*) é a capacidade de compreender "*o comportamento das aplicações executadas em ambientes complexos, (...) para entender o desempenho das aplicações, dos serviços e da infraestrutura, além de rastrear e responder a problemas tanto em tempo real quanto historicamente*" (ELASTIC, c2024). Você sabe que um sistema é observável quando você analisa os dados monitorados e consegue identificar e chegar a um diagnóstico. Comparando a um carro, as informações do painel — velocidade, temperatura, nível de combustível — possibilitam entender como o carro está funcionando internamente e identificar quando ocorre alguma anomalia.

Pilares da observabilidade

Para se obter um sistema observável, é preciso coletar e monitorar dados de **telemetria**. A Elastic (c2024) divide esse tipo de dado em três categorias, conhecidos como os ***pilares da observabilidade*** :

1. **Logs**: registra detalhes de forma cronológica de cada evento — o que aconteceu, quando ocorreu e tudo o que estava envolvido no momento. Essas informações fornecem contexto e são úteis para "debugar" código, explorar os problemas para chegar à causa raiz, ou, para investigar atividades suspeitas e ataques à segurança.

2. **Métricas**: valores numéricos mensurados em determinado intervalo de tempo. As métricas evidenciam o que precisa de atenção imediata e ajudam a identificar oportunidades de otimização.

3. **Rastreamento**: registra o caminho que uma requisição percorre em um sistema distribuído — serviços e componentes —, o que ajuda a identificar o código que está causando problemas, entender padrões de tráfego e as relações de dependência.

Isolados, os dados de telemetria não proporcionam uma visão completa do sistema, como aponta a Elastic (c2024), o rastreamento fornece "*visibilidade da integridade geral de uma aplicação, mas visibilidade limitada de sua infraestrutura subjacente. Para obter uma visão completa do seu ambiente, você precisa dos outros dois pilares da observabilidade: logs e métricas*".

Você pode monitorar inúmeras coisas, mas que não trazem clareza sobre o comportamento do sistema. Observabilidade se adquire quando se monitora as métricas certas, aquelas que refletem os fatores mais importantes da plataforma e ajudam a "*correlacionar tendências de performance e uso da API com as principais métricas*

de negócios, como receita e adoção, a fim de manter o alinhamento entre estratégias de negócio e técnicas" (POSTMAN, c2024).

Assim, os autores do livro *API Strategy for Decision Makers* (AMUNDSEN; GILLING, 2022) segmentam as métricas em **três tipos de observabilidades**:

1. **Máquina**: métricas de infraestrutura como CPU, memória, velocidade de disco indicam a integridade;
2. **Serviço**: métricas de processamento como leitura e gravação de dados no banco e requisições de cada *endpoint* indicam o que pode afetar a operação e, por consequência, o negócio;
3. **Produto**: métricas que indicam problemas que afetam diretamente o negócio, como pedidos concluídos, novos usuários e receita.

As métricas de máquina e serviço são relacionadas aos pilares da observabilidade e você as conheceu no capítulo *Métricas*, na Parte 2, quando a API é síncrona. Mas aquilo não é tudo. A depender da tecnologia, será necessário selecionar outras métricas. Alguns exemplos: sobre API assíncrona é importante saber o processamento de eventos por segundo, mensagens acumuladas em fila, entre outras; sobre infraestrutura Kubernets é relevante saber a quantidade de pods em funcionamento.

Já as de produto você vai selecionar aquelas que a sua plataforma suporta. Para a Order Platform, o número de pedidos é a métrica de negócio principal. Quando ocorre algo atípico, reflete nesse número instantaneamente.

Alerta

Produtos que derivam da plataforma só funcionam se as APIs funcionarem adequadamente. Sem o devido monitoramento, períodos de inatividade podem passar despercebidos, mas os usuários finais perceberão ao tentarem usar os apps. Por isso, o

monitoramento deve ser em tempo de execução e que dispare um alerta assim que alguma coisa atípica for identificada.

As métricas podem ser usadas para definir as regras de **alertas** que notificam o time da plataforma e, assim, agir proativamente antes que algo preocupante escale. Por exemplo, você pode definir que dispare um alerta quando atingir 10 respostas com erro 500, no intervalo de 1 minuto.

Application Performance Monitoring (APM)

Existem ferramentas de monitoramento de API, as **Application Performance Monitoring** (APM), que coletam os dados de telemetria e os tornam visíveis por meio de *dashboards* pré-construídos ou customizados de acordo com a necessidade.

APMs monitoram desde dependências até transações e a taxa de transferência (*throughput*) entre aplicações. Plataformas de API e *gateways* costumam fornecer este tipo de ferramenta, a exemplo do CloudWatch, para quem usa o AWS API Gateway. Basta configurar as métricas e os alertas. Há também opções *open source* como o Prometheus, além dos independentes Datadog, Sentry e New Relic.

É possível criar dashboards personalizados no Grafana que sintetizam os dados mais importantes de diferentes fontes, exclusivo para você e seu time.

Quem é responsável pelo monitoramento?

Quando você lança um serviço significa que você e seu time são responsáveis por mantê-lo ativo e funcionando. Vale a pena, para você como PM, acompanhar as métricas de saúde e aprender o padrão enquanto as coisas estão bem, assim, qualquer anormalidade vai chamar sua atenção. Mas essa atividade faz parte das suas responsabilidades? Depende da política da empresa.

No iFood, essa responsabilidade é das pessoas de engenharia do time, PMs não são cobrados por monitorar, nem para ficar disponível

em caso de incidente, como em plantões. Os times têm a liberdade de se auto-organizarem como acharem melhor. Como não há necessidade de várias pessoas fazerem a mesma coisa ao mesmo tempo, na Order Platform há dois papeis: um para monitorar durante o horário comercial, a pessoa "*hero*" (herói/heroína) do dia; e outra para plantão semanal fora do horário de trabalho.

A pessoa *hero* observa os gráficos de tempos em tempos e quando algo sai um pouco fora do normal, ela investiga e comunica ao time. Se a pessoa não conseguir identificar sozinha, ou a coisa se agrave nesse meio tempo, ela já aciona o time e uma ou mais pessoas param o que estão fazendo para ajudar — a depender do potencial de criticidade. Veja que isso acontece antes mesmo do alarme disparar. Claro que a pessoa *hero* do dia não faz somente isso, por essa razão os inúmeros alertas ajudam a garantir que o time seja notificado automaticamente quando a plataforma apresenta comportamentos fora do padrão.

Você continuaria usando um software que deixa você na mão? Não, né? Então, não há dúvida de que problemas de engenharia comprometem a retenção de clientes. Plataforma é um produto técnico, isto significa que algumas métricas de engenharia são as métricas do produto. Além de analisar o sucesso de lançamento de *features*, analisar as métricas de engenharia é uma forma de identificar oportunidades de melhoria e de orientar o planejamento de futuras iterações, mesmo que sejam puramente técnicas, afinal, existe o risco de perder clientes internos e usuários finais.

19.4 Evolução da plataforma

Como PM de produto que roda no browser, você pode lançar melhorias na interface (UX/UI) todos os dias, se quiser. O pior que pode acontecer é os usuários se perderem no fluxo que estavam acostumados e isso desagradá-los. Caso alguma funcionalidade

"quebre", pode ser corrigida rapidamente. Você pode lançar novas funcionalidades ou removê-las, e os usuários não precisam fazer nada para usufruir a versão mais recente.

Às vezes, mudanças em apps mobile exigem que o usuário atualize para a nova versão. Se as atualizações automáticas não estiverem habilitadas, será necessária uma ação manual do usuário para concluir essa atualização — um mero toque na tela, esforço praticamente zero.

O mesmo não se pode dizer sobre mudanças em APIs que já estão em uso. Se você lançar uma mudança no contrato de interface a cada mês, os consumidores da API precisarão adaptar a aplicação deles a cada publicação. Isso se traduz em excesso de esforço, de múltiplos times, apenas para manter a *feature* funcionando corretamente nos apps deles. Considerando as várias dependências, alterações em APIs custam caro. Isso está longe do que a empresa espera, assim como, não é essa experiência que você quer proporcionar aos times clientes, concorda?

Perceba o contraste com a expectativa de prototipagem rápida ou teste pequeno. Uma vez que a API está em uso, alterações podem interromper sistemas clientes e provocar o caos na empresa. É o tipo de coisa que pressiona o time da plataforma a acertar o design da API na primeira tentativa, exigência completamente diferente da realidade do design visual. "*Ainda não está claro como o estilo 'fail fast, fail often' (falhe rápido, falhe com frequência) do desenvolvimento ágil pode ser adaptado para a criação e evolução de APIs (...)*" (MYERS; STYLOS, 2015; tradução livre).

Isso significa que não se pode mudar uma API, nunca? Não é bem assim. Diferente de projeto com começo, meio e fim, API é um produto que certamente precisará evoluir para atender às necessidades do negócio ou de engenharia.

Lembra que foi mencionado antes que é possível integrar a uma API mesmo que o back-end não esteja pronto? É isso mesmo. Pode-se mudar a implementação (back-end), não a interface (o contrato). A implementação pode mudar com a frequência que você quiser, a API não. O time pode corrigir bugs, reduzir latência ou até reescrever todo o código na implementação. Mudanças que não alteram o contrato ficam ocultas atrás da interface e os times clientes se beneficiam das melhorias automaticamente. Isso não quer dizer que eles não serão afetados. Otimização na performance reflete nos apps deles e o efeito é percebido pelo usuário final.

Para que isso aconteça, o design da API deve ser elaborado com vistas a longo prazo, que atue como uma camada de proteção que dá a liberdade de alterar o back-end, como trocar o banco de dados, escalar a plataforma e refatorar, sem interromper as aplicações integradas à plataforma. Entretanto, software também envelhece e, em algum momento, você vai substituir uma API ou uma plataforma inteira por uma nova versão e descontinuar a anterior.

Dificilmente os times clientes vão reclamar de lançamentos que não alteram ou excluem funcionalidades existentes. Mas se eles precisarem alterar algo, ou pior ainda, migrar para uma nova versão porque a atual será descontinuada, prepare-se para enfrentar resistência. Os times não ficarão felizes em refazer o trabalho de integrar na nova versão para substituir algo que está funcionando para eles. Se você precisar quebrar esse tipo de resistência, *comece pelo plano de migração*, como explorado na seção anterior.

Concentrar-se em eficácia durante o *discovery* e eficiência durante o *delivery* facilita a adoção no lançamento. A retenção é alcançada quando a plataforma atende às necessidades dos times e do negócio com qualidade. O monitoramento é precioso, pois oferece visibilidade sobre o uso e saúde da plataforma, permitindo tomar decisões estratégicas para aprimorá-la. Como PMs, nossa missão é garantir que cada etapa do processo contribua para alcançar os objetivos estratégicos da empresa.

Ao longo dessa jornada, há desafios atípicos. Normalmente, gestão de produtos não se restringe ao produto, ela se expande a vários fatores que giram em torno dele. Esse aspecto é ainda mais evidente na gestão de plataformas, o que pode ser intensamente desafiador para PMs acostumados a produtos *customer-facing*. Prepare-se para um exame mais profundo no capítulo final, em que você vai compreender as razões por trás desses desafios e o seu papel nessa história toda.

Capítulo 20
Juntando tudo

Empresas adotam estratégia de plataforma para acelerar *time-to-market* e impulsionar a transformação digital — como se plataformas tivessem o poder natural de transformação. A verdade é que, assim como qualquer outra tecnologia, plataformas são o que se faz delas e, nesse contexto, significa que "*são tão eficazes quanto a estratégia de API empresarial que as orienta*" (AXWAY, c2021; tradução livre).

Construir plataformas e disponibilizar APIs não é o suficiente. É preciso ter uma visão clara do que se quer alcançar e como as APIs se encaixam nesse objetivo. Sem uma estratégia bem definida — tanto da empresa como da plataforma — e a abordagem adequada para guiar a construção, as APIs podem se tornar apenas mais um componente técnico, sem gerar valor real para o negócio.

A abordagem tradicional apoia-se no método projetizado, em que o objetivo é entregar um conjunto específico de funcionalidades dentro do prazo, sem abertura para melhorias futuras. As limitações dessa abordagem tornam APIs difíceis de se manter e expandir, comprometem a flexibilidade e a reutilização. É o que tem levado empresas a dificuldades para alcançar o crescimento esperado.

A natureza dinâmica dos negócios exige adaptações contínuas, portanto as plataformas precisam de atualizações para atender aos novos casos de uso. Isso pede uma mentalidade de constante evolução, com ciclos de vida prolongados e comprometimento com manutenção, atualização e inovação, fatores contemplados na abordagem de produto.

20.1 Plataforma como produto

Uma das razões pelas quais empresas não obtêm resultados da estratégia de plataforma é a ótica de que se trata de uma demanda técnica com a convicção de que é "só fazer". Não precisa de pesquisa, nem entender as necessidades dos usuários e contexto em que as coisas acontecem. Experiência do dev? Adoção? Para quê? Podemos simplesmente tornar a plataforma obrigatória em toda a empresa!

Pense um pouco comigo. Ninguém força você a comprar celular X. Você escolhe comprar, porque acredita que vale a pena. Só tem lógica obrigar a adesão de qualquer coisa por questões regulatórias ou de conformidade com a segurança. Portanto, se a utilização da sua plataforma não é por escolha, é sinal de que não vale a pena. Faz sentido?

Na perspectiva de produto, tudo se resume aos usuários e necessidades deles. Então, tratar plataformas como produto denota que "*tem clientes a serem conquistados, com uma proposta de valor, um roadmap claro e recursos dedicados para ajudar esses* `[times]` *clientes a gerarem valor para a organização. Assim como qualquer produto, uma plataforma precisa de atenção contínua, precisa evoluir e se adaptar em resposta ao feedback do developer* `[usuário]` *e ao panorama de negócio em mudança*" (LAYCOCK, c2024; Tradução livre).

Sob essa ótica, plataformas deixam de ser apontadas como uma solução técnica e passam a ser encaradas como um desafio interno, que atende às necessidades do negócio e contribui na alavanca dos KPIs. Isso requer adaptação de práticas de gestão de produtos na gestão de plataformas, em que observações além da tecnologia são incorporadas ao desenvolvimento da plataforma com intuito de oferecer valor, garantir aderência às necessidades dos times usuários e ao uso amigável (*dev friendly*), bem como evolução de acordo com as mudanças; isso tudo associado aos avanços tecnológicos.

Além de evitar a construção de um amontoado de coisas que ninguém precisa, essas observações servem para empacotar os

recursos em ofertas que os clientes internos desejam usar nas aplicações que desenvolvem. As práticas de produto ajudam a explorar o problema dos times internos, envolver os devs usuários, obter feedback e adequar a plataforma ao propósito. Trata-se de fazer o que é necessário para facilitar a vida dos devs e, assim, eles possam entregar experiências em forma de apps/produtos para os usuários finais.

Ao longo deste livro você viu a adaptação de algumas práticas de produto à gestão de plataforma (interna) que pretendem apoiar você na construção de plataformas que resolvem problemas de negócios e criam valor para a organização, assim como qualquer outro produto da empresa na qual você trabalha. Agora, é só pôr a mão na massa, a jornada será suave.

Não, infelizmente não é verdade. O livro orienta o que fazer e como fazer, mas gestão de plataforma carrega desafios que extrapolam (muito) o espaço da plataforma em si.

20.2 Plataforma é diferente de produto

Plataformas possuem características indissociáveis que tornam a gestão mais desafiadora do que gestão de apps/produtos. O mais interessante é que tais características parecem manifestar os mesmos desafios em todas as empresas, em qualquer tempo.

Para você ter uma ideia, uma dissertação sobre a principal plataforma interna da Nokia Networks, a DX 200 Platform, criada no período de 1996 a 1999, na unidade de pesquisa e desenvolvimento Switching Platforms (SWP), destaca que uma série de problemas ocorre devido à natureza complexa inerente aos ecossistemas de plataformas. Vejamos alguns deles.

Conflito entre estratégias de curto e longo prazo

O conflito entre a necessidade de resultados imediatos e o planejamento de longo prazo marcou o desenvolvimento da DX 200 Platform. As linhas de produto pressionavam para inserção de *features* de curto prazo.

A unidade SWP estava focada no desenvolvimento sustentável de longo prazo e chegavam demandas das estratégias de curto prazo das linhas de produtos tarde demais, o que dificultava a inclusão dessas demandas no planejamento da plataforma. Apesar da comunicação tardia, a decisão estratégica priorizava demandas de curto prazo, portanto o que estava planejado para ser desenvolvido era interrompido para que demandas *ad-hoc* de *features* fossem absorvidas pelo time da plataforma. No longo prazo, isso prejudicou a sustentabilidade e a capacidade de adaptar a plataforma às mudanças, resultando em perda de oportunidades.

Necessidades variadas e planejamento efêmero

Plataformas atendem a necessidades variadas, de estratégias diferentes. Por isso, vários stakeholders precisavam ser ouvidos. E, como é de se esperar, o estudo mostra que, quanto mais as necessidades são variadas — por vezes conflitantes —, mais exige adaptação da estratégica no desenvolvimento da plataforma para corresponder às necessidades.

Diferente de clientes externos, os internos estão em uma posição privilegiada (e em estado vigilante, eu diria) para questionar decisões e acompanhar de perto o progresso do desenvolvimento. E como se não bastasse, eles têm poder de influenciar o planejamento estratégico e a definição da plataforma, afinal, eles devem conseguir criar os apps deles e, indiretamente, vender a plataforma como parte integrante do produto para os clientes finais.

Processo de decisão burocrático

Planejamento de plataforma requer atividade transversal, apoiada por funções de controle também transversais, como grupos de governança ou comitês que lidam com necessidades conflitantes. A interface entre os times da plataforma e das linhas de produtos foi coordenada por sete comitês (*steering groups*), compostas inclusive por pessoas da alta gestão. Além da sobreposição de papéis e responsabilidades, o processo ficou complexo e burocrático, com reuniões grandes, dificuldades de priorização e ineficiências nas tomadas de decisão.

Priorização influenciada pelo poder

O contexto complexo revelou conflitos internos e politicagem que favoreciam os interesses dos stakeholders com mais poder. Muitas vezes, isso levava a decisões em que as linhas de produtos maiores ganhavam prioridade no desenvolvimento da plataforma, enquanto as menores eram negligenciadas.

Quando surgiam desacordos sobre as prioridades nos comitês, as decisões eram escaladas, às vezes chegando ao nível mais alto, a pessoa que centralizava a responsabilidade por todas as partes em discussão.

A influência decorrente do poder e do tamanho da linha de produtos moldou a estratégia da plataforma, no fim das contas.

Dependências e prazos rígidos

A priorização em meio ao desenvolvimento paralelo de plataformas e produtos derivados é complexa. A coordenação da cadência das entregas exige mais quantidade e qualidade na comunicação. Os produtos derivados dependem do progresso e conclusão das *features* na plataforma. Adivinha o que os stakeholders da DX 200 Platform mais queriam. Rigor dos cronogramas (prazo, como sempre!) e informação rápida e precisa em caso de alterações.

A pesquisa com os clientes internos revelou que "*cumprir os prazos era mais importantes do que lançamentos completos*" (RATAMÄKI, 2004). Como dizem, "feito é melhor do que perfeito". É compreensível, uma vez que as linhas de produto dependiam do cronograma para planejar seus próprios ciclos de desenvolvimento e atrasos ou falhas na comunicação impactam o lançamento.

Ausência de métricas

Foram estabelecidas e monitoradas métricas operacionais por diferentes comitês: recursos, portfólio, projetos, mudanças, qualidade do produto, capacidade de resposta aos clientes e satisfação do cliente. Porém, as métricas não eram específicas da DX 200 Platform, como também não foram definidas métricas estratégicas para avaliar a eficácia e a eficiência da plataforma.

A dissertação identifica a ausência de pensamento orientado ao mercado como a causa dessa lacuna nas métricas. Como uma plataforma interna sem alternativas externas, o time da plataforma não era pressionado para demonstrar valor aos clientes internos ou evidências de eficácia e eficiência. Essa desconexão entre as necessidades das linhas de produtos e a orientação estratégica do desenvolvimento da plataforma resultou em uma plataforma que não atendia plenamente às necessidades.

Tecnocentrismo sem visão de negócio

As estratégias de plataforma eram voltadas para tecnologia, o que resultou em desconexão entre as *features* desenvolvidas e as demandas de mercado. Os times da SWP sofreram críticas por se concentrarem na parte técnica, pela falta de conhecimento de negócio e distância dos clientes finais.

Com o objetivo de alinhar o desenvolvimento com as estratégias do negócio e as necessidades dos clientes finais, passou a ser requisitada a participação dos devs da plataforma nas reuniões das linhas de produto.

Features genéricas demais

A DX 200 Platform passou por um longo processo de produtização ao ser desacoplada de um produto existente. A primeira release envolveu a definição das interfaces juntamente com os times dos produtos derivados. Mas as discussões sobre a delimitação do que pertencia à plataforma e do que pertencia aos produtos derivados, sobre o design das interfaces, usabilidade e documentação, continuaram muito além do lançamento inicial, inclusive no nível estratégico. São preocupações pertinentes — afinal o trabalho de vários devs usuários é afetado por definições ruins — que evidenciam a complexidade do ambiente e o processo pesado e custoso.

Apesar de a base tecnológica compartilhada proporcionar otimização do uso de recursos, a DX 200 Platform falhava em atender adequadamente às demandas específicas de cada linha de produto — tensão clássica entre generalização e especialização. *Features* elaboradas para serem genéricas não atendiam completamente as necessidades de ninguém, ou, só atendiam à linhas de produtos específicas, utilizava recursos desnecessariamente e só ocupava espaço. O que deveria otimizar recursos, no fim, gerou uso ineficiente dos recursos.

E aí, parece com a sua rotina? Se esses eventos não aconteceram com você, calma, acontecerá. É surpreendente que o caso da dissertação tenha ocorrido há mais de 20 anos e esses padrões ainda sejam vistos nas empresas atualmente. Lidar com isso requer mudanças na forma de trabalhar e pensar que podem demorar um tempo para serem incorporadas.

20.3 Plataformize-se

No mundo da gestão de produtos, os fundamentos são os mesmos há décadas: entender o problema e quem passa por ele, idealizar soluções, implementá-las, coletar feedbacks e aprimorar continuamente. O tempo e esforço dedicados à gestão, discovery e delivery dependem de o produto ser tão simples quanto um aviãozinho de papel ou tão sofisticado quanto um foguete que pretende chegar até Marte. A maneira de evitar sofrimentos é incorporar a mentalidade de plataforma, se "autoplataformizando".

De interfaces visuais para capacidades

Para começar, é importante se desligar da mentalidade de construção de *interfaces visuais* (UI) e abraçar a ideia de que você está construindo *interfaces para capacidades do negócio*. Isso vai além de uma experiência específica em parte do app. São *features* que podem ser reutilizadas em diferentes contextos, idealizadas em módulos escaláveis em produtos derivados, que permitem combinações criativas e customizadas para atender às necessidades de diferentes clientes internos, bem como à demanda crescente e a futuras necessidades da organização. É absolutamente estratégico.

De produto individual para ecossistema

Decisões de design e desenvolvimento de uma plataforma não se limitam à estrutura dela. Elas impactam todo o ecossistema na qual ela está inserida, por isso, além de saber como a plataforma é estruturada, quais tecnologias são utilizadas e como as diferentes partes se integram, PMs de plataforma precisam de visão holística para compreender a integração com outras plataformas, apps, produtos até chegar na experiência dos clientes finais.

A habilidade de comunicação é fundamental para qualquer PM, e no ambiente complexo de ecossistema o trabalho é redobrado. Naturalmente, o techniquês fará mais parte do seu vocabulário. Você passará a compreender e se comunicar melhor com as pessoas técnicas. Já com outros perfis de stakeholders é preciso saber traduzir termos técnicos em termos de negócio, abstrair a

complexidade em explicações simples. No geral, você terá de entender o que a tecnologia traz de valor e usar isso como argumento das escolhas. E assim como em times de qualquer produto, você pode pedir apoio das pessoas de engenharia para apontar e explicar as vantagens e desvantagens.

De habilidades comuns à específicas

PMs são responsáveis por entender as necessidades dos clientes e criar um produto para atendê-las. Fazemos isso empregando habilidades comuns de gestão de produtos. Além das habilidades de produto, PMs de plataforma necessitam de um certo nível a mais de conhecimento técnico do que PMs de produto *customer-facing*, porque alguns dos problemas são relacionados à escala e ao design de arquitetura.

O primeiro passo é saber jogar com as *skills* que você carrega dos fundamentos da gestão de produtos e, com o tempo, absorver habilidades específicas que ajudem na gestão de plataformas, como noções de arquitetura, tipos de APIs e formas de integração com outros sistemas e, assim, adquirir a capacidade de construir soluções escaláveis que atendam a vários times e sustentem o crescimento em longo prazo.

Além de atender a necessidades variadas, as habilidades específicas permitirão que você veja além dos problemas imediatos até ampliar para um olhar holístico do ecossistema ao qual a sua plataforma pertence. Foi o que eu fiz, e parte disso resultou neste livro.

De curto a longo prazo

Uma diferença que a gente percebe de cara: experimentos podem ser uma péssima ideia em plataformas que já são base de produtos derivados. Ganhos rápidos (*quick wins*), tão comuns no desenvolvimento de produtos *customer-facing*, podem prejudicar a longevidade da plataforma, por isso, dificilmente serão bem-vindos. Implementações de longo prazo, embora essenciais, são difíceis de

conquistar *buy-in* e, mesmo com todos comprados, basta surgir qualquer iniciativa com suposto potencial de resultados no curto prazo que o plano de longo prazo sofre interferências. Isso mostra quão frágil é manter o compromisso por tanto tempo.

Como convencer stakeholders a priorizar necessidades do futuro, que nem existem ainda, em detrimento de necessidades do agora?! Raramente isso é possível, então garanta que os stakeholders tenham total ciência dos *trade-offs* e implicações entre *quick wins* e investimentos de longo prazo. O alinhamento das expectativas precisa ficar claro, transparente como cristal.

Incorporar a mentalidade de plataforma vai além de conquistar habilidades técnicas. Exige o desenvolvimento uma nova forma de pensar e agir em concordância com a mentalidade de plataforma. E não para por aí. A estratégia de plataforma bem-sucedida depende do quanto a empresa está aderente a essa mentalidade também.

20.4 Mudanças na empresa

Engana-se quem pensa que passar por transformação digital ou aderir à estratégia de plataforma se limita a uma mudança tecnológica. No momento da decisão, a maioria das pessoas não têm noção de que esse tipo de mudança permeia toda a organização, impacta desde processos até modelos de negócios e pessoas em todos os níveis da organização.

É natural que mudanças dessa magnitude, que movimentam processos, estruturas de times e até cultura organizacional, causem fortíssima resistência. Toda a liderança tem papel fundamental na condução da mudança, não apenas a liderança técnica. É essencial que compreendam e apoiem a estratégia de plataforma. Mesmo com todo o apoio, essa jornada é repleta de desentendimentos, como os vistos sobre a DX 200 Platform.

A dissertação sobre a unidade SWP da Nokia Networks traz dois pontos que chamam atenção. Uma delas é que a perda de oportunidades e o prejuízo à sustentabilidade da DX 200 Platform, resultante da priorização de demandas de curto prazo, provocaram mudanças no direcionamento estratégico: a visão de longo prazo da plataforma ganhou prioridade. A outra é que, apesar das queixas, a pesquisa com os stakeholders indicou que os comitês contribuíram para a coerência das decisões, alinhadas com os objetivos estratégicos de longo prazo da organização.

Isso mostra que nem tudo o que parece ruim faz mal ao todo, entretanto algumas coisas são pura ilusão, que induzem a expectativas improváveis e precisam de um choque de realidade.

Mudança de paradigma

A expectativa de desenvolvimento e resultados rápidos é incompatível com a natureza complexa das plataformas e o ecossistema envolvido. Essa divergência entre expectativa e realidade gera cobranças incabíveis sobre os times de plataforma, que muitas vezes enfrentam pressão para entregas de resultados rápidos, mesmo que isso comprometa a sustentabilidade e escalabilidade da plataforma a longo prazo.

A conscientização sobre a distinção entre construção de plataformas e produtos *customer-facing* é o caminho para promover a mudança de paradigma na empresa. É preciso que os stakeholders compreendam a diferença e entendam a importância de uma visão de longo prazo que priorize escalabilidade, flexibilidade e adaptabilidade da plataforma para que o negócio possa reagir com agilidade às mudanças do mercado.

Essa mudança de mentalidade muda a forma como a empresa opera e toma decisões. Por exemplo, a cadência trimestral pode ser adequada para cumprir o planejado para apps, enquanto, para plataformas, três meses é um período muito curto para que algo

relevante esteja pronto. Portanto, será que não faz mais sentido o planejamento semestral para plataformas? Fica aí uma provocação!

O pensamento de produto e o pensamento de plataforma se complementam. As plataformas fornecem a base tecnológica robusta e flexível em longo prazo para que os times de produto tenham a capacidade de explorar novas oportunidades, iterar e responder às demandas do mercado com agilidade. Dessa forma, plataformas, junto aos produtos derivados, impulsionam o crescimento sustentável e competitividade do negócio.

Mudança na estratégia

Como ilhas isoladas, algumas plataformas (senão todas) são ignoradas no plano estratégico organizacional. Algumas são lembradas, mas sem um objetivo estratégico claro que as conecte aos objetivos de negócio. Outras são tão descoladas da estratégia da empresa, que o time da plataforma sequer vê valor no que está construindo — mais perceptível quando as plataformas são orientadas a projetos.

É válido acompanhar o progresso do desenvolvimento de plataformas como projetos, especialmente quando há muitas dependências, mas o objetivo ainda deve ser a entrega de algo que se manifeste no negócio e contribua nos principais KPIs da empresa de alguma forma. Contudo, tão prejudicial quanto ignorar as plataformas no planejamento estratégico é forçar aderência das iniciativas de plataforma aos objetivos de negócio. Essa abordagem, apesar da boa intenção, geralmente é disfuncional, como a imposição de metas da empresa incompatíveis com o que a plataforma é capaz de executar — exemplo visto no capítulo *Métricas*, em que o OKR compartilhado entre times era a taxa de conversão via *push notification*.

Assim como qualquer outro produto, espera-se que as métricas de sucesso sejam coerentes com a capacidade da plataforma. Se isso não é possível, em vez de excluir plataformas do mapa estratégico,

será que não seria mais sensato incorporá-las à estratégia geral da empresa como uma nova frente, com um papel claramente definido e indicadores alinhados às capacidades delas? Mais uma provocação, aqui.

Uma frente estratégica para plataformas — a exemplo clássico da Volkswagen — redefine a percepção sobre elas. Eleva-as de ferramentas tecnológicas para ferramentas estratégicas, ativos de crescimento e inovação para a empresa. Entretanto, a estratégia de plataforma é alinhada às tendências e demandas futuras do mercado. Nesse contexto, assume-se um caráter proativo, no sentido de se antecipar às situações, em que se define o que será feito agora para garantir que as plataformas apoiem tanto apps atuais quanto os futuros.

Essa visão vai muito além de apenas reagir às condições imediatas do mercado. Ela prepara a organização para se adaptar às condições que surgirão, de modo que evolui junto às necessidades do negócio e garante suporte ao crescimento da empresa em longo prazo.

Produtize, plataformize, evangelize

Você, como PM de plataforma, tem a oportunidade de atuar como agente de mudança. Como? Você produtiza quando insere e propaga práticas de produto no dia a dia de desenvolvimento da plataforma; quando entrevista times clientes; quando promove workshops de design thinking para colaboração entre times e stakeholders; quando adapta os princípios de agilidade ao contexto de plataformas; e mesmo que em períodos mais longos, quando almeja entrega de valor contínua, você **evangeliza sobre produtização** em um ambiente notavelmente técnico.

Todas as vezes em que você interage com outros times, transita em outras áreas, participa de reuniões com a alta gestão, você tem a oportunidade de **evangelizar sobre plataformização** ao expor conceitos de plataforma, ao questionar o que parece desalinhado com esses conceitos ou equívocos na interpretação; ao propor

outras formas de trabalho, de inserir plataformas no mapa estratégico, de sugerir indicadores coerentes com a natureza de plataformas.

A adoção da estratégia de plataforma é o ponto de partida para as empresas se reinventarem e PMs de plataforma têm a contribuir com isso. Nessa jornada, estamos todos aprendendo sobre produtização de plataformas. Para PMs sem background técnico, é um caminho árduo que pede reinvenção da forma de fazer gestão de produto. Este livro surge como um guia para suavizar algumas passagens íngremes, auxiliando você a exercer a famosa "influência sem autoridade" para propagar a mentalidade de produto entre os times de plataformas e a mentalidade de plataforma em outras áreas e níveis hierárquicos. Essa é uma missão a conta-gotas que demanda disposição e perseverança.

Sabendo disso, você escolhe se reinventar como PM? A verdadeira transformação só ocorrerá se você acreditar no poder das plataformas para impulsionar o negócio. Para evangelizar outros, é preciso ser evangelizado(a) primeiro. Este livro cumpriu esse papel para você? Se a sua jornada for semelhante à minha, prepare-se para dias difíceis, mas saiba que superar o desafio será surpreendentemente recompensador.

Capítulo 21
Referências

ALECRIM, Emerson. Uber tem prejuízo de US$ 1,2 bilhão, mas número de viagens aumenta. *Tecnoblog*, 2022. Disponível em: https://tecnoblog.net/noticias/2022/11/01/uber-tem-prejuizo-de-us-12-bilhao-mas-numero-de-viagens-aumenta/. Acesso em: 4 mar. 2023.

AMUNDSEN, Mike; GILLING, Derric. **API Strategy for Decision Makers: Growing Your Product Through Customer Observability**. United States of America: O'Reilly Media, Inc., 2022.

ANDREESSEN, Marc. The only thing that matters. *Pmarchive*, 2007. Disponível em: https://pmarchive.com/guide_to_startups_part4.html. Acesso em: 27 nov. 2022.

APIGEE. Web API Design: The Missing Link. Google Cloud. 2018. Disponível em: https://cloud.google.com/static/files/apigee/apigee-web-api-design-the-missing-link-ebook.pdf. Acesso em: 2 dez. 2022

APIARY. How to Build an API. c2022. Disponvível em:

https://apiary.io/how-to-build-api#phase-preparation. Acesso em: 24 out 2022.

AWS. *O que são microsserviços*. In: Microsserviços. c2022. Disponível em: https://aws.amazon.com/pt/microservices/. Acesso em: 18 jul. 2022.

AWS. **Amazon Cognito anuncia Acordo de nível de serviço de 99,9%**. 2019. Disponível em: https://aws.amazon.com/pt/about-aws/whats-new/2019/01/amazon-cognito-announces-99-9-service-level-agreement/. Acesso em: 9 mar. 2023.

AWS. **Preço do Amazon API Gateway**. c2022. Disponível em: https://aws.amazon.com/pt/api-gateway/pricing/. Acesso em: 9 mar. 2023

AWS. O que é uma API RESTful?. c2024. Disponível em: https://aws.amazon.com/pt/what-is/restful-api/. Acesso em: 23 jun. 2024.

AXWAY. **Are you getting the maximum value from your APIs?**. c2021. Disponível em: https://resources.axway.com/reuse-apis-tech/guide-enterprise-api-strategy. Acesso em: 21 jan 2021.

AXWAY. *Forrester Opportunity Snapshot: Closing the Gap Between API Development and API Consumption*. In: API Management. c2022. Disponível em: https://resources.axway.com/api-management-doc/report-forrester-closing-the-gap-between-api-development-and-api-consumption. Acesso em: 20 jan. 2023.

AZURE ARCHITECTURE CENTER. Scalable order processing. *Microsoft Learn*, c2023. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/example-scenario/data/ecommerce-order-processing. Acesso em: 27 jul. 2022.

BECK, Kent; BEEDLE, Mike; BENNEKUM, Arie van; COCKBURN, Alistair; CUNNINGHAM, Ward; FOWLER, Martin; GRENNING, James; HIGHSMITH, Jim; HUNT, Andrew; JEFFRIES, Ron; KERN, Jon; MARICK, Brian; MARTIN, Robert C.; MELLOR, Steve; SCHWABER, Ken; SUTHERLAND, Jeff; THOMAS, Dave. Manifesto para Desenvolvimento Ágil de Software. 2001. Disponível em: https://agilemanifesto.org/iso/ptbr/manifesto.html. Acesso em: 5 abr. 2023

Big data. In: Wikipedia, the free encyclopedia.

Disponível em: https://pt.wikipedia.org/wiki/Big_data. Acesso em: 05 jul 2024.

BLOOMBERG. Amazon's Website Crashed Within Minutes of Starting Amazon Prime Day and the Internet Is Losing It. *Money*, 2018. Disponível em: https://money.com/amazons-website-crashed-within-minutes-of-starting-amazon-prime-day-and-the-internet-is-losing-it/. Acesso em: 5 dez. 2022.

CAGAN, Marty; JONES, Chris. **Empowered**: Ordinary People, Extraordinary Products. Hoboken, New Jersey: John Wiley & Sons, Inc., 2021.

CAGAN, Marty. **Product Management – Start Here**. Silicon Valley Product Group. 2020. Disponível em: https://www.svpg.com/product-management-start-here . Acesso em: 16 jun. 2022.

CAGAN, Marty. **Product vs Feature Teams**. Silicon Valley Product Group. 2019. Disponível em: https://www.svpg.com/product-management-start-here . Acesso em: 29 jun. 2024.

CLOUD ELEMENTS. **The State of API Integration 2021 report**. 2021. Disponível em:

https://f.hubspotusercontent40.net/hubfs/440197/Cloud-Elements-2021-SOAI.pdf. Acesso em: 1 dez. 2022.

COCHRAN, Tim. Maximizing Developer Effectiveness. *martinfowler.com*, 2021. Disponível em: https://martinfowler.com/articles/developer-effectiveness.html. Acesso em: 8 jun. 2021.

COLUCCI, Lucas. Quanto a empresa perde financeiramente quando o projeto atrasa?. *Plataformatec*, 2017. Disponível em: https://blog.plataformatec.com.br/2017/06/quanto-a-empresa-perde-financeiramente-quando-o-projeto-atrasa/. Acesso em: 7 mar. 2023.

CRAWFORD, Merle; DI BENEDETTO, Anthony. *Gestão de novos produtos*. 11. ed. Porto Alegre: AMGH, 2016.

CONTENTSTACK. **Monolithic vs. Microservices: Why Decoupled and Headless Architectures are the Future**. c2023. Disponível em: https://www.contentstack.com/cms-guides/monolithic-vs-microservices-cms-architectures/. Acesso em: 4 dez. 2022.

DESHPANDE, Amit; SINGH, Nampreet P. Challenges and patterns for modernizing a monolithic application into microservices. *IBM Developer*. 2021. Disponível em: https://developer.ibm.com/articles/challenges-and-patterns-for-modernizing-a-monolithic-application-into-microservices/. Acesso em: 11 dez. 2022.

DRUCKER, Peter F.; Managing for Business Effectiveness. **Harvard Business Review**. 1963. Disponível em: https://hbr.org/1963/05/managing-for-business-effectiveness. Acesso em: 5 abr. 2023

EASTWOOD, Brian. What makes a product platform succeed?. *MIT Sloan School of Management*, 2016. Disponível em: https://mitsloan.mit.edu/ideas-made-to-matter/what-makes-a-product-platform-succeed. Acesso em: 7 nov. 2022

ELASTIC. **O que é o OpenTelemetry?**. c2024. Disponível em: https://www.elastic.co/pt/what-is/opentelemetry. Acesso em: 10 mar 2024.

FRAZIER, Ross; KHAN, Naufal; LUNAWAT, Gautam; RAHUL, Amit. Products and platforms: Is your technology operating model ready?. *McKinsey Digital*, 28 fev. 2020. Disponível em:

https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/products-and-platforms-is-your-technology-operating-model-ready. Acesso em: 31 maio 2021.

FILE: Neil Mcelroy's 1931 Brand Man Memo.pdf. In: *Wikipedia, The Free encyclopedia*. Disponível em: https://en.wikipedia.org/w/index.php?title=File:Neil_Mcelroy%27s_1931_Brand_Man_Memo.pdf&page=3. Acesso em: 14 jan. 2023.

GILLING, Derric. What does API Monitoring Mean for API Product Managers and Growth Teams. *Moesif*, 2022. Disponível em: https://www.moesif.com/blog/api-product-management/api-metrics/What-Does-API-Monitoring-Mean-For-API-Product-Managers-and-Growth-Teams/. Acesso em: 25 fev. 2023.

GOOGLE CLOUD. *The API Product Mindset*: How to move fast, delight customers, and continually innovate to thrive in today's economy. 2018. Disponível em: https://cloud.google.com/files/apigee/apigee-api-product-mindset-ebook.pdf. Acesso em: 15 jul. 2022.

GOOGLE DEVELOPERS. Content API for Shopping.

Orders resource. c2021. Disponível em: https://developers.google.com/shopping-content/guides/orders-guide. Acesso em: 17 ago. 2021.

GOOGLE DEVELOPERS. Content API for Shopping. **Sobre pedidos**. c2023. Disponível em: https://developers.google.com/shopping-content/guides/about-orders?hl=pt-br. Acesso em: 20 jun. 2023.

HANSSON, David Heinemeier. We spent $3,201,564.24 on cloud in 2022 at @37signals, mostly AWS. $907,837.83 on S3. $473,196.30 on RDS. $519,959.60 on OpenSearch. $123,852.30 on Elasticache. This is with long commits (S3 for 4 years!!), reserved instances, etc. Just obscene. Will publish full accounting soon. 12 de jan. 2023. Twitter: @dhh. Disponível em: https://twitter.com/dhh/status/1613508201953038337. Acesso em: 12 de jan. 2023.

HARLAND, Peter E.; UDDIN, Zakir; LAUDIEN, Sven. Product platforms as a lever of competitive advantage on a company-wide level: a resource management perspective. *Review of Managerial Science*, v. 14, p. 137-158, 2020. Disponível em:

https://doi.org/10.1007/s11846-018-0289-9. Acesso em: 10 jun. 2022.

HARRIS, Chandler. Microservices vs. monolithic architecture. *Atlassian*, c2022. Disponível em: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. Acesso em: 4 dez. 2022.

HASSO PLATTNER INSTITUTE OF DESIGN AT STANFORD. *An Introduction to Design Thinking*: Process Guide. c2023. Disponível em: https://s3-eu-west-1.amazonaws.com/ih-materials/uploads/Introduction-to-design-thinking.pdf. Acesso em: 11 maio 2023.

HUNTER, Kirsten L. Irresistible APIs: Designing web APIs that developers will love. Shelter Island, NY: Manning Publications, 2017.

IDEO. Design thinking defined. c2023. Disponível em: https://designthinking.ideo.com/. Acesso em: 11 maio 2023.

IDEO. How do people define design thinking?. c2023. Disponível em: https://designthinking.ideo.com/faq/how-do-people-

define-design-thinking. Acesso em: 12 maio 2023.

INTERACTION DESIGN FOUNDATION. What is Design Thinking?. c2023. Disponível em: https://www.interaction-design.org/literature/topics/design-thinking. Acesso em: 11 maio 2023.

IRIGARAY, Hélio A. R.; VIANNA, Alexandre; NASSER, José E.; LIMA, Luiz P. M. *Gestão e desenvolvimento de produtos e marcas*. 3. ed. Rio de Janeiro: Editora FGV, 2011.

JACOBSON, Daniel; BRAIL, Greg; WOODS, Dan. **APIs: A Strategy Guide**. Canada: O'Reilly Media, Inc., 2011.

JOHNSON, Tom. **API design and usability**. idratherbewriting.com. 2019. Disponível em: https://idratherbewriting.com/learnapidoc/evaluating-api-design.html. Acesso em: 09 maio 2023

JUSTICE, Lorraine; HAREL, Dan. Design Thinking Fundamentals. [Curso online]. Rochester Institute of Technology, c2019. Disponível em: https://www.edx.org/course/design-thinking-fundamentals. Acesso em: 16 fev. 2019.

KEMPF, Rachel. Monolithic Applications vs. Microservices. *Azion*, 2021. Disponível em: https://www.azion.com/en/blog/monolithic-applications-vs-microservices/. Acesso em: 4 dez. 2022.

KOLOSSOVSKI, Eleanor. A Strategist's Guide to Platform Thinking. *Medium*, 2019. Disponível em: https://medium.com/@eleanor.kolossovski/a-strategists-guide-to-platform-thinking-ebf5837e2fa0. Acesso em: 10 jun. 2022.

KOTLER, Philip; KELLER, Kevin L. **Administração de marketing**. 15. ed. São Paulo: Pearson, 2019.

LAYCOCK, Rachel. **Platforms as products**: Delivering value through customer focus. Insights in Thoughtworks. c2024. Disponível em: https://www.thoughtworks.com/en-ec/insights/looking-glass/platforms-as-products. Acesso em: 2 maio 2024.

LEWIS, James; FOWLER, Martin. Microservices. *martinfowler.com*, 2014. Disponível em: https://www.martinfowler.com/articles/microservices.html. Acesso em: 18 jul. 2022.

List of HTTP status codes. In: Wikipedia, the free encyclopedia. Disponível em: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes. Acesso em: 17 jul 2023.

MASSÉ, Nicolas. Full API lifecycle management: A primer. *Red Hat Developer*, 2019. Disponível em: https://developers.redhat.com/blog/2019/02/25/full-api-lifecycle-management-a-primer. Acesso em: 5 jan. 2023.

MAURYA, Ash. *Running Lean*: Iterate from Plan A to a Plan that Works. 3. ed. Canada: O'Reilly Media Inc., 2022.

MEDJAOUI, Mehdi; WILDE, Erik; MITRA, Ronnie; AMUNDSEN, Mike. *Continuous API Management*: Making the Right Decisions in an Evolving Landscape. 1. ed. United States of America: O'Reilly Media, Inc., 2018.

MIRO. Low-fidelity Wireframes Template. c2023. Disponível em: https://miro.com/templates/low-fidelity-wireframes/ Acesso em: 1 jul. 2023.

MIT SLOAN EXECUTIVE EDUCATION. Product

Platforms: A Source of Competitive Advantage webinar with Dr. Bruce Cameron. *YouTube*, 2015. Disponível em: https://youtu.be/sXyopYwbIx0. Acesso em: 6 jun. 2022.

MYERS, Brad A.; STYLOS, Jeffrey. **Improving API Usability**. 2015. Disponível em: https://www.cs.cmu.edu/~NatProg/papers/API_Usability_Article_submitted.pdf. Acesso em: 11 maio 2023.

NIELSEN, Jakob. **10 Usability Heuristics for User Interface Design**. Nielsen Norman Group. 2020. Disponível em: https://www.nngroup.com/articles/ten-usability-heuristics/. Acesso em: 11 maio 2023

Paralisia por análise. In: WIKIPÉDIA: a enciclopédia livre. Disponível em: https://pt.wikipedia.org/wiki/Paralisia_por_an%C3%A1lise. Acesso em: 29 maio 2023.

PATTON, Jeff. **Opportunity Canvas**. 2016. Disponível em: https://jpattonassociates.com/opportunity-canvas/. Acesso em: 16 nov. 2022.

PATTON, Jeff. **User Story Mapping**: Discover the Whole Story, Build the Right Product. United States of America: O'Reilly Media, Inc., 2014.

PICCIONI, Marco; FURIA, Carlo A.; MEYER, Bertrand. **An Empirical Study of API Usability**. c2023. Disponível em: https://se.inf.ethz.ch/~meyer/publications/empirical/API_usability.pdf. Acesso em: 11 maio 2023.

PM SCHOOL. All you need to know about APIs | Product Management | PM School. *YouTube*, 2022. Disponível em: https://youtu.be/T5y0WKmKenQ. Acesso em: 30 out. 2022.

Portal IPNews. *Brasil amarga quatro anos de atraso no mercado de computação em nuvem*. 2022. Disponível em: https://ipnews.com.br/brasil-amarga-quatro-anos-de-defasagem-no-mercado-de-computacao-em-nuvem/. Acesso em: 9 mar. 2023.

POSTMAN. **2022 State of the API Report**. 2022. Disponível em: https://www.postman.com/state-of-api/#key-findings. Acesso em: 2 dez. 2022.

POSTMAN. **API lifecycle**. c2024. Disponível em: https://www.postman.com/api-platform/api-

lifecycle/. Acesso em: 24 jun. 2024

POSTMAN. **API observability**. c2024. Disponível em: https://www.postman.com/api-platform/api-observability/. Acesso em: 09 mar. 2024

Programa Olá Mundo. In: *Wikipedia, The Free encyclopedia*. Disponível em: https://pt.wikipedia.org/wiki/Programa_Ol%C3%A1_Mundo. Acesso em: 29 dez. 2022.

RATAMÄKI, Katja. **Product Platform Development from the Product Lines' Perspective: Case of Switching Platform**. Lappeenranta, 2004. Disponível em: https://lutpub.lut.fi/bitstream/handle/10024/31193/TMP.objres.306.pdf. Acesso em: 11 maio 2024.

RIES, Eric. *A startup enxuta*: como os empreendedores atuais utilizam a inovação contínua para criar empresas extremamente bem-sucedidas. São Paulo: Lua de Papel, 2012.

SALESFORCE. *12 Benefits of Cloud Computing and Its Advantages*. c2023. Disponível em: https://www.salesforce.com/products/platform/best-practices/benefits-of-cloud-computing/. Acesso em: 7

fev. 2023.

SALINAS, Erica; LU, Jack. Scaling DLT to 1M TPS on AWS: Optimizing a Regulated Liabilities Network. **AWS Architecture Blog**. 2022. Disponível em: https://aws.amazon.com/pt/blogs/architecture/scaling-dlt-to-1m-tps-on-aws-optimizing-a-regulated-liabilities-network/. Acesso em: 22 jun. 2023

SAP. *What is a technology platform?*. In: SAP Insights. c2023. Disponível em: https://www.sap.com/insights/what-is-a-technology-platform.html. Acesso em: 7 fev. 2023.

SEGMENT. Plans scale with your growth. c2023. Disponível em: https://segment.com/pricing/. Acesso em: 10 mar. 2023

SELLERS, Amanda. How to Create Content for Every Stage of the Buyer's Journey. *HubSpot*, 2022. Disponível em: https://blog.hubspot.com/marketing/content-for-every-funnel-stage. Acesso em: 16 dez. 2022.

SQUARE DEVELOPER. *Orders API*. c2021. Disponível em: https://developer.squareup.com/docs/orders-api/what-it-does. Acesso em: 17 ago. 2021.

SWAGGER. *Swagger Petstore - OpenAPI 3.0*. c2023. Disponível em: https://petstore3.swagger.io/. Acesso em: 29 abr 2023

THURSTON, David. Using Twilio Functions to Add Custom JavaScript Code in Studio Flows. Twilio Blog. 2018. Disponível em: https://www.twilio.com/en-us/blog/custom-javascript-twilio-functions-code-studio-flows-html. Acesso em: 29 jun. 2024.

TORRES, Teresa. **Why this opportunity solution tree is changing the way product teams work**. Product Talk. 2016. Disponível em: https://www.producttalk.org/2016/08/opportunity-solution-tree/. Acesso em: 27 jul. 2022

VALE, Alisson. **A Fórmula da Eficácia**: como fazer a coisa certa no seu projeto de software. Brasília, DF: Software Zen, 2020.

VALE, Alisson. A arte da gestão de software. Curso online. Software Zen, c2021. Disponível em: https://softwarezen.com.br/produtos/software-zen/. Acesso em: 4 out. 2021.

WATKINS, Michael. *The First 90 Days*: proven

strategies for getting up to speed faster and smarter. Massachusetts: Harvard Business Review Press, 2013.